UNIVERSITE BRETAGNE LOIRE / MATHSTIC

UBO
Université de Bretagne Occidentale

THÈSE DE DOCTORAT DE

L'UNIVERSITÉ
DE BRETAGNE OCCIDENTALE
COMUE UNIVERSITÉ BRETAGNE LOIRE

ECOLE DOCTORALE N° 602
Mathématiques et Sciences et Technologies de l'Information et de la Communication
Spécialité : informatique

Par

Steven COSTIOU

Adaptation non-anticipée de comportement : application au déverminage de programmes en cours d'exécution

Thèse présentée et soutenue à Brest, le 28 novembre 2018

Unité de recherche : Lab-STICC UMR CNRS 6285

Composition du Jury :

Présidente	
Isabelle BORNE	Professeure des universités, Université de Bretagne Sud
Rapporteurs	
Luc FABRESSE	Professeur, IMT Lille Douai, Institut Mines-Telecom
Christophe DONY	Professeur des universités, Université de Montpellier
Examinateurs	
Stéphane DUCASSE	Directeur de recherche, INRIA Lille - Nord Europe
Directeur de thèse	
Alain PLANTEC	Professeur des universités, Université de Bretagne Occidentale
Encadrant	
Mickaël KERBOEUF	Maître de conférences, Université de Bretagne Occidentale
Invité	
Loïc LAGADEC	Professeur, ENSTA Bretagne

Remerciements

Je remercie mon directeur de thèse Alain Plantec ainsi que Mickaël Kerboeuf pour leur encadrement durant la thèse, à la fois pour la grande autonomie qu'ils m'ont laissé et leur présence indispensable et bienveillante.

Je tiens à remercier chaleureusement les membres du jury : Isabelle Borne pour la présidence du jury, Luc Fabresse et Christophe Dony pour avoir accepté de rapporter la thèse et leurs retours élogieux et constructifs, ainsi que Stéphane Ducasse et Loïc Lagadec.

Je remercie les nombreuses personnes de l'université avec qui j'ai eu l'occasion de collaborer tout au long de la thèse : les membres du département informatique, du Lab-STICC, du Continuum Lycées-Universités et de l'Open Factory.

Je souhaite remercier de manière particulière Loïc Lagadec, dont les cours d'ingénierie logicielle et de Smalltalk, il y a presque 15 ans maintenant, ont été déterminants dans le chemin que j'ai parcouru depuis, Stéphane Ducasse qui m'a poussé à publier à des moments clés de ma thèse, Emanuel Radoi qui m'a permis de m'investir dans le laboratoire et ses aspects humains, Catherine Archieri, Anne Lagadec et Hélène Couthon du Continuum, qui m'ont donné une perspective unique sur le monde de l'orientation et son animation, les membres de l'équipe RMoD, en particulier Anne, Damien, Marcus, Vincent et Christopher pour leur aide précieuse avant ma soutenance, ainsi que l'équipe de Diazol et Éric et Pierre de Thales.

Je remercie l'ensemble de mes amis et de ma famille qui m'ont soutenu dans cette aventure. Je pense à Valérie, et à nos longues et passionnantes discussions sur l'enseignement et la transmission du savoir aux enfants. Je pense à Stéphane, Maël, Thomas et Younn, avec qui j'ai partagé une partie de ma jeunesse et qui, chacun à leur manière, continuent à m'inspirer. À Glenn, compagnon d'infortune, désormais c'est ton tour de soutenir. Aux gens du #chan, qui se reconnaitront : Thomas, Arnaud, Yann, Julien, Jessica, Antoine et Z. Enfin, je dédie cette thèse à ma famille, à ma mère et mon père, à ma sœur Stéphanie et mon frère Kévin, à Jean-Jacques et Claire et à ma très très chère cousine Héloïse, et surtout à Clotilde, ma compagne, qui a été d'un soutien et d'une patience infinis durant ces trois années.

Table des matières

Table des matières **v**

Table des figures **xi**

Liste des tableaux **xv**

Introduction **1**

I État de l'art et de la pratique 7

1 Déverminage : un état de la pratique **9**
1.1 Enjeux et difficultés du déverminage 10
1.1.1 Généralités 10
1.1.2 Une classification des *bugs* difficiles 11
1.1.3 Outillage du déverminage 13
1.2 Méthodes, techniques et outils pour le déverminage 13
1.2.1 Techniques de base 13
1.2.2 Techniques avancées 17
1.3 Difficultés spécifiques du déverminage en cours d'exécution 23
1.3.1 Déverminage de programmes en cours d'exécution 23
1.3.2 Déverminage centré objet 24
1.3.3 Proposition 25

2 Adaptation non-anticipée de comportement dans les programmes à objets : un état de l'art **29**
2.1 Mise à jour et reconfiguration de programmes en cours d'exécution . . 31
2.1.1 Mise à jour à chaud 31
2.1.2 *Live Programming* 32
2.1.3 Systèmes à base de composants 34
2.1.4 Discussion 37
2.2 Paradigmes permettant la variation comportementale 38

2.2.1 La Programmation Orientée Aspects (AOP) 38
2.2.2 La programmation par les Rôles 41
2.2.3 Programmation Orientée Contexte (COP) 46
2.2.4 Discussion 49
2.3 Approches réflexives 51
2.3.1 Classes anonymes, *wrappers*, *proxies* 52
2.3.2 Protocoles à méta-objets et méta-objets 55
2.3.3 Discussion 61
2.4 Infrastructures dédiées à l'adaptation 62
2.4.1 *Chisel* 62
2.4.2 *LyRT* 63
2.4.3 Discussion. 64
2.5 Synthèse et conclusion 65

II Patron de langage objet dynamique pour l'adaptation non-anticipée de comportement 69

3 Patron de langage objet pour l'adaptation non-anticipée de comportement 71
3.1 Kernel-Lub : un langage objet minimal 73
3.1.1 Infrastructure de Kernel-Lub 73
3.1.2 Opérations du langage 74
3.1.3 Modèle d'exécution 75
3.1.4 Sémantique opérationnelle 76
3.2 Lub : une extension pour l'adaptation non-anticipée de comportement 77
3.2.1 Infrastructure de Lub 78
3.2.2 Opérations du langage 78
3.2.3 Modèle d'exécution 79
3.2.4 Sémantique opérationnelle 80
3.2.5 Adaptations gardées 81
3.2.6 Conception d'adaptation comportementale avec Lub 82
3.2.7 Impact de l'adaptation sur le lookup 83
3.3 Collecteurs : groupes d'objets dynamiques pour l'adaptation collective 87
3.3.1 Infrastructure des Collecteurs 87
3.3.2 Opérations du langage 88
3.3.3 Modèle d'exécution 89
3.3.4 Sémantique opérationnelle 89
3.3.5 La collecte dynamique d'objets : une illustration 92
3.4 Propriétés fondamentales 94
3.4.1 Kernel-Lub : formalisation du lookup 94

3.4.2 Lub : formalisation du lookup 98
3.5 Conclusion 102

III Mise en œuvre du patron de langage et application au déverminage non-anticipé 103

4 Mise en oeuvre et évaluation du patron de langage pour l'adaptation de comportement non-anticipée : Lub 105
4.1 Principes de mise en œuvre de Lub dans un langage existant 108
4.1.1 Proposition de mise en œuvre 108
4.1.2 Avantages de la solution de mise en œuvre proposée 110
4.1.3 Inconvénients de la solution proposée 111
4.1.4 Mise en œuvre concrète au travers d'une *API* 113
4.2 Mise en œuvre dans Pharo avec Reflectivity 122
4.2.1 Reflectivity : la couche réflexive de Pharo 122
4.2.2 Une extension de Reflectivity pour une granularité objet . . . 124
4.2.3 Bénéfices de l'extension de Reflectivity 129
4.2.4 Modèle de mise en œuvre de Lub basé sur Reflectivity 130
4.2.5 Interfaces pour l'adaptation 132
4.3 Application du patron de langage : une mise en œuvre en Python avec les *Talents* 134
4.4 Analyse des performances 135
4.4.1 Critères de comparaison 135
4.4.2 Protocole de la prise de mesure 136
4.4.3 Périmètre des mesures 136
4.4.4 Évaluation de la vitesse du lookup instrumenté 137
4.4.5 Évaluation de la vitesse d'adaptation des objets 138
4.4.6 Évaluation de la consommation mémoire des objets adaptés . 140
4.4.7 Limites de l'évaluation des performances et de la consommation mémoire 142
4.5 Adaptation non-anticipée de comportement dans une flotte de drones 143
4.5.1 Scénario : la perte de GPS 144
4.5.2 Stratégie de déverminage : l'adaptation non-anticipée de comportement 145
4.5.3 Description de la simulation 145
4.5.4 Adaptations au fil de l'eau 146
4.5.5 Discussion 153
4.6 Évaluation 155
4.7 Conclusion 156

5 Mise en œuvre du patron de langage pour la collecte d'objets non-anticipée : les Collecteurs 159
5.1 Contraintes et options de mise en œuvre des Collecteurs 161
5.1.1 Contraintes de mise en œuvre 161
5.1.2 Options de mise en œuvre pour l'application au déverminage . 163
5.1.3 *Metalinks* dédiés pour la collecte d'objets 165
5.1.4 Collecte d'objets *en profondeur* 166
5.2 Mise en œuvre concrète des Collecteurs 166
5.2.1 Modèle de mise en œuvre avec Pharo et Reflectivity 166
5.2.2 Limites du modèle 172
5.2.3 L'API des Collecteurs 172
5.2.4 Mise en œuvre avec Python et Reflectivity 176
5.3 Outils pour l'application au déverminage 177
5.3.1 Outillage des Collecteurs 178
5.3.2 Visualisation des objets collectés : les historiques de collecte . 181
5.4 Traces d'objets non-anticipées dans un programme en cours d'exécution 184
5.4.1 Description du problème 184
5.4.2 Traces dynamiques d'objets collectés 185
5.4.3 Discussion 185
5.5 *Record/Replay* pour la reproduction de valeurs non-déterministes . . 186
5.5.1 Description du problème 186
5.5.2 Rejouer une exécution avec un objet collecté 187
5.5.3 Discussion 191
5.6 Historiques de collecte pour l'investigation de problèmes difficiles . . . 192
5.6.1 Description du problème 193
5.6.2 Observation du programme avant la modification problématique 194
5.6.3 Observation du programme après introduction de l'erreur . . . 195
5.6.4 Traque du problème et correction 196
5.6.5 Discussion 199
5.7 Discussion générale et évaluation 200
5.7.1 Évaluation 200
5.7.2 Discussion 201
5.8 Conclusion 202

6 Une infrastructure pour l'adaptation non-anticipée de comportement appliquée au déverminage d'applications en cours d'exécution 205
6.1 Infrastructure et outillage pour l'adaptation non-anticipée dans les programmes en cours d'exécution 207
6.1.1 *Debug-Scopes* : périmètres dynamiques de l'adaptation 208
6.1.2 Définition d'un périmètre, de ses activateurs et de ses collecteurs 212
6.1.3 Définition d'adaptation de comportement 213

6.1.4 Visualisation du comportement adapté 217
6.1.5 Compatibilité avec les outils natifs de Pharo 218
6.2 Cas d'utilisation sur un objet connecté : l'application *Sensor Monitoring App* et le *chat-bot IotZ* 218
6.3 Scénario 1 : déverminage non-anticipé en cours d'exécution 221
6.3.1 Détection du problème 222
6.3.2 Correction du problème par une adaptation de comportement 223
6.3.3 Discussion 226
6.4 Scénario 2 : utilisation non-anticipée de la caméra pour la prise de photographies de surveillance 227
6.4.1 Adaptation du programme pour utiliser la caméra 227
6.4.2 Discussion 233
6.5 Scénario 3 : extension d'interface utilisateur pour du déverminage . . 234
6.5.1 Adaptation de l'interface du *chat-bot* 235
6.5.2 Discussion 238
6.6 Discussion générale et évaluation 239
6.6.1 Évaluation 239
6.6.2 Bénéfices des périmètres d'adaptation 240
6.6.3 Problèmes et limitations 240
6.7 Conclusion 243

Conclusion **245**

Bibliographie **255**

Table des figures

2.1 Illustration d'un système à composants 35
2.2 Principe de l'AOP 39
2.3 Illustration de traces d'exécution mises en œuvre avec un aspect 40
2.4 Illustration de deux instances jouant des rôles 42

3.1 Infrastructure de *Kernel-Lub* 73
3.2 Opérations du langage minimal 74
3.3 Modèle d'exécution minimal 75
3.4 Infrastructure de Lub : *Kernel-Lub* étendu avec Adaptation 78
3.5 Opérateurs de Lub étendus avec Adaptation 79
3.6 Modèle d'exécution de Lub étendu avec Adaptation 79
3.7 Lub : infrastructure pour adaptations gardées 82
3.8 Impact de l'adaptation sur le *lookup* 85
3.9 Infrastructure de Lub étendue aux Collecteurs 87
3.10 Opérations de Lub étendues aux Collecteurs 89
3.11 Programme illustratif pour la collecte d'objets. 92
3.12 Illustration d'un collecteur 93
3.13 Espaces de noms 95
3.14 Définition de *graphe de programme* 95
3.15 Propriété de l'héritage simple 95
3.16 Définition de *graphe de système d'objets* 96
3.17 Fonctions de *lookup* 97
3.18 Propriétés du *lookup* 97
3.19 Noms d'adaptations 98
3.20 Graphe de programme adaptable 98
3.21 Validité des adaptations 99
3.22 Système d'objets adaptable 99
3.23 Fonction de *lookup* adapté 99
3.24 Propriétés du *lookup* adaptable 100
3.25 Non réduction d'interface 100
3.26 Démonstration de la propriété de non réduction d'interface 101

4.1 Adaptation d'un objet (illustration) . . . 108
4.2 Adaptation d'un objet (mise en œuvre) . . . 109
4.3 Problème du super . . . 112
4.4 Adaptateur d'objets . . . 114
4.5 Interface de l'adaptateur d'objets . . . 115
4.6 Gestionnaire de sous-classes anonymes . . . 117
4.7 Algorithme de génération de sous-classe anonyme . . . 118
4.8 Migrateur d'objets . . . 119
4.9 Algorithmes de migration d'objets . . . 120
4.10 Illustrations d'un *metalink* . . . 123
4.11 Interface de l'extension de Reflectivity . . . 126
4.12 Algorithmes de gestion de metalinks centrés objets suivant les principes généraux de Lub . . . 127
4.13 Saut de *lookup* . . . 128
4.14 Mise en œuvre du saut de *lookup* avec Reflectivity . . . 129
4.15 Configuration d'un *metalink* pour l'adaptation d'une méthode . . . 132
4.16 Lub, API de définition d'adaptations. . . . 133
4.17 Lub, illustration d'une adaptation . . . 133
4.18 Lub, API d'adaptation d'objets. . . . 134
4.19 Scénario d'adaptation dans une simulation de drones . . . 144
4.20 Extrait du modèle de la classe *LSDrone* . . . 147
4.21 Extrait des traces d'exécution reçues par l'opérateur-rice . . . 148
4.22 Adaptation développée pour surveiller les messages reçus . . . 149
4.23 Spécification d'une adaptation pour un port de communication d'un drone 150
4.24 Traces d'exécution après adaptation . . . 150
4.25 Définition de l'adaptation de la méthode *handleMessage*: dans une nouvelle classe . . . 151
4.26 Adaptation de la méthode *handleMessage*: du drone *follower-drone* . . . 151
4.27 Résultat de l'exécution après adaptation . . . 152
4.28 Suppression de l'adaptation du port de communication drone à drone de *follower-drone* . . . 153

5.1 Illustration d'un collecteur . . . 161
5.2 Expression dont le résultat de l'exécution doit être collecté . . . 162
5.3 Ré-écriture de code source pour l'installation d'un collecteur . . . 163
5.4 Collection en profondeur . . . 167
5.5 *Metalink* de collecte . . . 168
5.6 Comportement de collecte implémenté par la classe *CollectBehavior*. . . . 169
5.7 Modèle des collecteurs. . . . 171
5.8 Menu de définition de collecteurs à partir du code source. . . . 178
5.9 Visualisation des points de collecte dans le code source. . . . 179

5.10 Configuration d'un collecteur . . . 179
5.11 Sélection des réifications . . . 180
5.12 Intégration des Collecteurs dans l'inspecteur natif de Pharo . . . 181
5.13 Vue d'ensemble d'un inspecteur ouvert sur un point de collecte . . . 182
5.14 Détail d'un historique de point de collecte . . . 183
5.15 Problème non-déterministe, extrait de [Schulz and Bockisch, 2017] . . . 187
5.16 Configuration d'une collecte conditionnelle . . . 188
5.17 Visualisation d'une collecte . . . 189
5.18 Visualisation d'une expression dont l'exécution est remplacée par un objet fixe . . . 190
5.19 Restriction d'une opération de collecte et de *replay* à un objet en utilisant l'inspecteur de Pharo. . . . 190
5.20 Point de collecte sur l'appel à l'accesseur *disabledPhases* . . . 195
5.21 Variation des objets collectés (montage) . . . 195
5.22 Méthode appelée dans la pile d'exécution de la première configuration collectée . . . 196
5.23 Pile d'exécution amenant à la collecte de la seconde configuration instanciée 197
5.24 *Lookup* des propriétés des configurations Pillar . . . 198
5.25 Correction expérimentée à la suite de la création d'une nouvelle sous-configuration . . . 198

6.1 Navigateur de périmètres . . . 207
6.2 Modèle de mise en œuvre des *Debug-Scopes*. . . . 210
6.3 Adaptation d'un programme distant . . . 211
6.4 Définition d'un activateur par interaction directe avec le code source. . . . 212
6.5 Spécification d'un activateur conditionnel . . . 213
6.6 La spécification d'adaptation pour un collecteur . . . 214
6.7 Menu contextuel pour la définition d'un contrôle d'adaptation, pour la méthode d'adaptation sélectionnée. . . . 215
6.8 Définition du contrôle d'une méthode adaptée . . . 216
6.9 Réifications du contexte d'exécution d'une méthode . . . 216
6.10 Visualisation du comportement adapté pour un périmètre d'adaptation . 217
6.11 L'application connectée *Sensor Monitoring App* en fonctionnement . . . 219
6.12 Extrait de données de surveillance affichées dans la discussion. . . . 220
6.13 L'application connectée *Sensor Monitoring App* avec son *chat-bot IotZ*. . 221
6.14 Extrait de données de surveillance problématiques affichées sur le serveur de discussion. . . . 222
6.15 Activateurs : montage de la Figure 6.1 . . . 225
6.16 Adaptation de comportement appliquée à la méthode $convertRawValue$ des instances collectées . . . 225
6.17 Activation du périmètre *NaN Fix* . . . 226

6.18 Définition d'un point de collecte sur la variable *self* 228
6.19 Définition d'une adaptation qui utilise la caméra du *bot* 228
6.20 Méthodes de publication sur le serveur de discussion (classe *IotApp*). . . 229
6.21 Configuration du contrôle de l'adaptation 229
6.22 Visualisation du contrôle de l'adaptation 230
6.23 Garde d'adaptation sous forme d'activateur conditionnel 231
6.24 Garde d'adaptation sous forme d'activateur basé sur le flot de contrôle . 231
6.25 Traces d'exécution sous forme de photographie 232
6.26 Définition de l'adaptation d'extension de l'interface du bot 235
6.27 Une nouvelle méthode adapte et remplace la méthode *keywords* 236
6.28 Compatibilité du comportement adapté avec le dévermineur Pharo . . . 236
6.29 Capture d'écran du serveur de discussion Discord 237

Liste des tableaux

2.1 Positionnement des approches de type mise à jour à chaud et reconfiguration dynamique . . . 37
2.2 Positionnement des approches de type paradigmes de programmation pour l'adaptation non-anticipée . . . 50
2.3 Positionnement des approches réflexives à base de restructuration et d'encapsulation . . . 55
2.4 Positionnement des approches réflexives basées sur les protocoles à méta-objets . . . 61
2.5 Positionnement des infrastructures dédiées à l'adaptation non-anticipée . 64
2.6 Synthèse du positionnement des approches abordant le minage d'objet . 65
2.7 Synthèse du positionnement des approches étudiées pour l'adaptation non-anticipée . . . 67

3.1 Résultats du *lookup* avec et sans adaptation . . . 85

4.1 Scénarios d'évaluation du lookup . . . 137
4.2 Comparaison des vitesses d'exécution du lookup pour 10^6 exécutions (en millisecondes) . . . 138
4.3 Temps d'exécution de l'adaptation d'objets (en millisecondes) . . . 139
4.4 Temps d'exécution de la restauration d'objets à leur comportement initial (en millisecondes) . . . 140
4.5 Comparaison de la consommation mémoire de 1000 objets adaptés par Lub avec 1000 objets adaptés par héritage dans Pharo (en octets). . . . 142
4.6 Positionnement de Lub et de sa mise en œuvre . . . 156

5.1 Positionnement des Collecteurs et de leur mise en œuvre . . . 200

6.1 Positionnement des *Debug-Scopes* et de leur mise en œuvre . . . 239

Introduction

Le déverminage est l'activité de recherche, d'identification et de correction de problèmes dans un programme. Il s'agit d'une activité difficile et chronophage. Cette investigation passe par différentes phases : l'observation du problème et sa correction. Lorsque le problème est difficile, il est parfois nécessaire de le reproduire et de l'observer plusieurs fois pour en comprendre la source. Ces opérations impliquent traditionnellement de stopper le programme, d'instrumenter ce dernier pour reproduire le problème, de corriger le programme puis de mettre à jour son code source avant de le redéployer.

Problème

Certains problèmes n'apparaissent pas avant que le programme ne rencontre un certain contexte d'exécution. D'autres problèmes n'apparaissent que dans un environnement de production, et ne peuvent pas être reproduits dans un environnement de développement. D'autres encore sont dus à des causes non-déterministes, et empêchent leur observation dans des environnements contrôlés comme les suites de tests unitaires. Cela rend très difficile la reproduction de ces problèmes, qui est essentielle pour comprendre leurs causes et pouvoir concevoir un correctif.

De manière orthogonale, certains programmes ont la contrainte de devoir fonctionner en continu. Ils ne peuvent pas s'arrêter le temps d'une mise à jour − typiquement dans ce cas pour l'installation d'un correctif. C'est le cas des applications fournissant des services qui ne doivent pas être interrompus (banques, serveurs de discussion...), de programmes qui accumulent des états qui seraient perdus par un redémarrage (simulations), ou encore des systèmes cyber-physiques qui évoluent dans des environnements imprévisibles (robots, drones, objets connectés, satellites...).

Lorsque ces deux situations sont réunies, à savoir 1) l'accès au contexte d'exécution du programme au moment de l'apparition d'un problème est essentiel pour comprendre et résoudre ce dernier et 2) le programme ne peut pas être arrêté pour être corrigé, il devient primordial de pouvoir observer, instrumenter et corriger le programme *à chaud* pendant son exécution.

En outre, les problèmes qui surviennent en cours d'exécution − voire en production − sont par définition imprévus. Il est alors peu probable que le programme contienne déjà les instrumentations nécessaires pour investiguer et corriger ces problèmes. Nous dirons alors que l'activité de déverminage est non-anticipée, situation pour laquelle les techniques et les outils de débogage actuels atteignent rapidement leurs limites.

Contexte

Nous nous plaçons dans le contexte de la programmation objet. Dans ce cadre, une nouvelle difficulté s'ajoute à celles précédemment évoquées, qui est la granularité des entités déboguées. L'instrumentation d'un programme, pour l'observer ou le corriger, se fait principalement au travers des méthodes décrites dans les classes de ces programmes. Tous les objets instances des classes instrumentées sont alors impactés par l'instrumentation. Cependant, un besoin récurrent lors de l'investigation de problèmes difficiles est l'observation ou l'instrumentation du comportement de certains objets précis, indépendamment de leur classe. Une fois qu'un problème a été identifié, la modification du comportement du programme devrait idéalement pouvoir se faire avec une granularité objet pour faciliter l'expérimentation de correctifs très ciblés.

Problématiques de recherche

Nous nous intéressons donc au déverminage non-anticipé d'objets dans les programmes en cours d'exécution. Cela nous amène à l'étude de deux problématiques, qui sont les suivantes :

1. Comment peut-on déverminer des objets de manière non-anticipée, pendant l'exécution d'un programme ?
2. Comment spécifier et obtenir les ensembles d'objets à déverminer de manière non-anticipée, pendant l'exécution du programme ?

Ces questions se posent pour une vaste majorité de programmes orientés objet, mais qui peuvent reposer sur des langages avec des caractéristiques très hétérogènes (à typage statique ou dynamique, à typage fort ou faible, etc.). Les contraintes des programmes peuvent également fortement varier selon le domaine d'application visé (*temps-réel*, parallélisme, performances...). Dans cette thèse, nous restreignons le périmètre d'étude de ces problématiques aux programmes écrits avec des langages objet dynamiquement typés, sans contraintes de parallélisme ni de *temps-réel*.

Proposition

Pour obtenir la possibilité de déboguer des objets de manière flexible, c'est-à-dire avec des instrumentations librement spécifiées par les développeur-euse-s en fonction du problème rencontré par le programme, nous proposons d'adapter dynamiquement le comportement de ces objets. Plutôt que de reposer sur des infrastructures spécifiques, nous proposons de partir du langage initialement utilisé pour développer le programme, puis de l'étendre de manière légère et générique avec des capacités d'adaptation dynamique à granularité objet.

Contributions

Les contributions de cette thèse sont les suivantes :

1. La description d'un patron de langage objet dynamique, avec la capacité de spécifier des groupes d'objets à collecter dans le programme en cours d'exécution, et la capacité d'adapter le comportement de ces objets. Le patron de langage est constitué des éléments suivants :
   — *Kernel-Lub*, un modèle de langage objet minimal qui représente les langages dynamiques pouvant être étendus par le patron de langage,
   — *Lub*, un patron de langage pour l'adaptation non-anticipée d'objets,
   — Les *Collecteurs*, un patron de langage pour la spécification et le regroupement non-anticipés d'objets.
2. Une mise en œuvre de *Lub* et des *Collecteurs* sous la forme d'extensions des langages Pharo [Black et al., 2010] et Python [Python, 2017]. La mise en œuvre Pharo du patron est mise en pratique au travers d'exemples et sur des cas d'utilisation de déverminage.
3. Une mise en œuvre d'outils de déverminage dédiés reposant sur l'extension de Pharo par le patron de langage. Ces outils prennent la forme d'interfaces de programmation et utilisent les capacités d'adaptation d'objets apportées par *Lub* et les *Collecteurs* pour mettre en pratique des techniques de débogage
4. Une infrastructure dédiée au déverminage d'objets par l'adaptation non-anticipée de comportement, nommée *Debug-Scopes*, qui exploite la combinaison de *Lub* et les *Collecteurs*.

Structure de la dissertation

Cette thèse est divisée en trois parties et six chapitres, dont le contenu est détaillé ci-dessous.

Première partie. Dans le **chapitre 1**, nous étudions l'état de la pratique et des techniques de déverminage dans le cadre des problématiques abordées. Nous nous intéressons en particulier à la capacité et aux limites des outils à déboguer des objets de manière non-anticipée. Dans ce contexte, nous précisons notre proposition, ainsi que la notion d'adaptation non-anticipée de comportement, puis nous identifions sept propriétés désirables pour le déverminage d'objets par l'adaptation non-anticipée de comportement. Dans le **chapitre 2**, nous dressons un état de l'art des solutions d'adaptation de comportement non-anticipée, que nous évaluons au travers des sept propriétés recherchées pour le débogage d'objets dans les programmes en cours d'exécution.

Deuxième partie. Cette partie contient le **chapitre 3**, qui introduit et détaille la proposition centrale de la thèse. Il s'agit d'un patron de langage, comprenant un modèle de langage objet minimal ainsi que deux extensions, *Lub* pour l'adaptation non-anticipée d'objets et les *Collecteurs* pour la spécification dynamique de groupes d'objets à adapter.

Troisième partie. Cette partie contient la mise en œuvre et l'évaluation du patron de langage dans les langages Pharo et Python. Le **chapitre 4** décrit la mise en œuvre de *Lub*, concernant l'adaptation non-anticipée d'objets. Le **chapitre 5** décrit la mise en œuvre des *Collecteurs* et de leurs outils, concernant la spécification dynamique de groupes d'objets à soumettre à l'adaptation. Le **chapitre 6** décrit les *Debug-Scopes*, une infrastructure pour le déverminage non-anticipé d'objets basée sur l'extension du langage Pharo par *Lub* et les *Collecteurs*. Chacun de ces chapitres est illustré par des cas d'utilisation de débogage, et contient une évaluation des mises en œuvre présentées au regard des propriétés recherchées pour le déverminage d'objets non-anticipé.

Une conclusion clôture la thèse, et ouvre des perspectives de travaux futurs.

Publications

Les travaux décrits dans cette thèse ont fait l'objet de plusieurs publications. D'autres travaux, non référencés dans ce manuscrit mais cependant essentiels dans la réflexion menée et les directions choisies, ont également fait l'objet de publications. La liste complète de ces publications, au total 8 articles, est présentée ci-dessous.

1. COSTIOU, Steven, KERBOEUF, Mickaël, CAVARLÉ, Glenn and PLANTEC, Alain. Lub : A pattern for fine grained behavior adaptation at runtime. Science of Computer Programming, 2018, vol. 161, p. 149-171.

2. CAVARLÉ, Glenn, PLANTEC, Alain, COSTIOU, Steven and RIBAUD, Vincent. A feature-oriented model-driven engineering approach for the early validation of feature-based applications. Science of Computer Programming, 2018, vol. 161, p. 18-33.

3. COSTIOU, Steven, TOULLEC, Clotilde, KERBOEUF, Mickaël and PLANTEC, Alain. Back-in-time inspectors : an implementation with Collectors. In : Proceedings of the 13th edition of the International Workshop on Smalltalk Technologies. ACM, 2018.

4. COSTIOU, Steven, KERBOEUF, Mickaël, PLANTEC, Alain and DENKER, Marcus. Collectors. In : Conference Companion of the 2nd International Conference on Art, Science, and Engineering of Programming. ACM, 2018. p. 144-152.

5. MARRA, Matteo, BOIX, Elisa Gonzalez, COSTIOU, Steven, KERBOEUF, Mickaël, PLANTEC, Alain, POLITO, Guillermo and DUCASSE, Stéphane. Debugging Cyber-Physical Systems with Pharo : An Experience Report. In : Proceedings of the 12th edition of the International Workshop on Smalltalk Technologies. ACM, 2017. p. 8.

6. COSTIOU, Steven, KERBOEUF, Mickaël, DENKER, Marcus and PLANTEC, Alain. Unanticipated debugging with dynamic layers. In : Companion to the first International Conference on the Art, Science and Engineering of Programming. ACM, 2017. p. 14.

7. COSTIOU, Steven, KERBOEUF, Mickaël, CAVARLÉ, Glenn and PLANTEC, Alain. Lub : a dsl for dynamic context oriented programming. In : Proceedings of the 11th edition of the International Workshop on Smalltalk Technologies. ACM, 2016. p. 13.

8. CAVARLÉ, Glenn, PLANTEC, Alain, COSTIOU, Steven and RIBAUD, Vincent. Dynamic Round-Trip Engineering in the context of FOMDD. In : Proceedings of the 11th edition of the International Workshop on Smalltalk Technologies. ACM, 2016. p. 15.

Première partie

État de l'art et de la pratique

Cette partie contient deux chapitres. Dans le premier chapitre, un état de la pratique du déverminage décrit les techniques et les outils actuels pour observer, reproduire et corriger des problèmes, ainsi que leurs limites pour aborder les problèmes les plus difficiles. Parmi ces derniers, le débogage de programmes en cours d'exécution est un cas de figure particulièrement difficile à aborder avec les outils existants. La proposition d'utiliser l'adaptation non-anticipée de comportement pour déboguer les programmes est alors introduite. Dans le second chapitre, un état de l'art de l'adaptation de comportement non-anticipée dresse un panorama des possibilités existantes pour adapter des programmes en cours d'exécution. Les solutions étudiées sont positionnées au regard de propriétés désirables pour le débogage d'objets non-anticipé.

Chapitre 1

Déverminage : un état de la pratique

Sommaire

1.1 Enjeux et difficultés du déverminage 10
1.1.1 Généralités 10
1.1.2 Une classification des *bugs* difficiles 11
1.1.3 Outillage du déverminage 13
1.2 Méthodes, techniques et outils pour le déverminage 13
1.2.1 Techniques de base 13
1.2.2 Techniques avancées 17
1.3 Difficultés spécifiques du déverminage en cours d'exécution 23
1.3.1 Déverminage de programmes en cours d'exécution 23
1.3.2 Déverminage centré objet 24
1.3.3 Proposition 25

Le déverminage, ou débogage, est une activité difficile. Ce chapitre se focalise sur une difficulté particulière de l'activité de déverminage, qui concerne les problèmes dont les symptômes sont éloignés de leur source. Ces problème sont particulièrement difficiles à comprendre, et parfois à observer et à reproduire. Nous dressons un bref panorama des techniques de débogage, les plus courantes comme les plus avancées, et comment ces dernières tentent d'aborder ce genre de *bugs*. Enfin, nous mettons en avant les limites de l'état de la pratique du déverminage lorsque ces problèmes difficiles surviennent de manière non-anticipée, dans des applications devant fonctionner en continu et qu'il n'est pas possible d'interrompre pour investiguer. Pour pallier ces limites, nous formulons une proposition basée sur l'*adaptation non-anticipée de comportement* dans les programmes à objet. Cette proposition vise à instrumenter dynamiquement des objets avec des comportements de débogage, de manière non-anticipée et sans interrompre le fonctionnement du programme.

1.1 Enjeux et difficultés du déverminage

Le déverminage − ou débogage − de programmes est l'activité générique d'investigation et de correction de problèmes dans des programmes informatiques [Agans, 2002, Zeller, 2009]. Il s'agit en pratique de rechercher et de comprendre pourquoi un programme ne se comporte pas comme il le devrait, puis de le corriger [Agans, 2002]. Un problème peut prendre différentes formes [Zeller, 2009]. Un défaut est un code défectueux dans le programme, qui peut provoquer une infection (corruption de l'état du programme, comportements incohérents ou inattendus) et à terme provoquer une erreur, observable de manière externe au programme : le programme s'arrête, ou ne fonctionne plus correctement. Le débogage selon Zeller consiste alors à identifier et à caractériser la chaîne d'infection, à trouver et comprendre sa source puis à corriger le défaut qui la provoque, de manière à ce que l'erreur ne se reproduise plus. On parlera de *bug* pour désigner de manière générique un problème, quelle que soit sa caractérisation (défaut, infection, erreur).

1.1.1 Généralités

Le déverminage est une activité difficile, et cette difficulté est multifactorielle. Notamment, il existe plusieurs catégories de *bug*, avec des sources et des impacts différents. Par exemple d'après Eisenstadt [Eisenstadt, 1997], plus de 50% des difficultés proviennent de la distance entre la manifestation d'un *bug* et sa source − c'est-à-dire qu'il n'y a pas de relation évidente entre la source d'un *bug* et ses symptômes − ainsi que des problèmes qui rendent les outils inopérants ou inefficaces. Une erreur triviale peut alors être difficile à corriger parce qu'elle n'a pas de lien avec sa manifestation observable, ou encore parce que le code où se situe le défaut est considéré comme un code de confiance et n'est donc pas assez rapidement investigué [Knuth, 1989]. Cette difficulté liée à la distance source-symptôme d'un *bug* est toujours d'actualité aujourd'hui [Böhme et al., 2017, Perscheid et al., 2017]. Une autre catégorie de *bug* nommée *suprise scenarios* ou *higher level-surprises* [Knuth, 1989] englobe des cas particulièrement complexes liés à des interactions non-anticipées entre différentes parties du programme, ou à des erreurs *dormantes* qui ne font pas partie des cas d'utilisation les plus courants. Ces erreurs peuvent donc rester invisibles aux utilisateurs du programme, jusqu'à ce qu'un symptôme se déclare. Dans un autre registre, les erreurs liées aux défauts de conception ou au parallélisme font aujourd'hui partie des problèmes les plus durs à résoudre lorsqu'on interroge les développeurs et les développeuses [Layman et al., 2013, Perscheid et al., 2017]. Suivant la nature d'un problème, il peut être difficile de le reproduire, processus clé pour le comprendre et en trouver l'origine (section 1.1.2). D'autre part, le manque d'outils adéquats pour aborder un problème donné constitue également une limitation (section 1.1.3). Ces difficultés ralentissent et perturbent le processus de débogage.

1.1.2 Une classification des *bugs* difficiles

La première étape du déverminage est l'observation du programme. L'observation permet d'accumuler des informations sur un problème afin de pouvoir le reproduire [Agans, 2002]. La reproduction d'un problème consiste ensuite à déterminer et à isoler ses conditions d'apparition [Zeller, 2009]. Il s'agit de deux étapes systématiquement observées dans l'activité de débogage [Katz and Anderson, 1987, Agans, 2002, Zeller, 2009, Layman et al., 2013, Perscheid et al., 2017].

Observation du programme. L'objectif est de définir le contour du problème [Agans, 2002, Murphy et al., 2006, Zeller, 2009, Layman et al., 2013] et d'en extraire les caractéristiques afin de se construire une représentation mentale du programme en cours d'exécution. Ce modèle mental permet ensuite de choisir une stratégie de correction du *bug* [Katz and Anderson, 1987, Layman et al., 2013]. Lors de l'observation, il est souvent nécessaire de visualiser le programme de manière très large, puis de restreindre le périmètre de visualisation pour observer plus finement le comportement du programme et chercher à valider les hypothèses formulées [Agans, 2002, Zeller, 2009, Layman et al., 2013, Perscheid et al., 2017].

Reproduction du problème. Une fois que le problème a clairement pu être observé, il faut le comprendre. Cette étape de compréhension passe par une reproduction du *bug*, dans l'espoir d'en trouver la source. Il est parfois nécessaire de comparer une exécution reproduisant le *bug* de manière certaine avec une exécution dite *saine* du programme, pour accumuler assez d'informations et d'en déterminer les différences [Agans, 2002, Zeller, 2009, Layman et al., 2013].

Difficultés. Certains *bugs* sont concrets, faciles à détecter, à observer et à reproduire. Ils sont parfois dits *solides*, ou surnommés *Bohrbugs*, en référence à l'atome de Bohr [Gray, 1986]. Ces facilités d'observation et de reproduction ne permettent pas de présumer de la complexité de la résolution d'un *bug*, mais son périmètre est bien défini et un-e développeur-euse peut y travailler. Il en va tout autrement des problèmes qui sont difficiles à observer ou à reproduire. Les raisons de ces difficultés sont multiples, et ces *bugs* sont surnommés suivant tout autant d'analogies [Raymond and Steele, 1996] :

- Le *Heisenbug* [Gray, 1986, Eisenstadt, 1997, Agans, 2002, Zeller, 2009], qui disparaît lorsqu'on tente de l'observer et rend inefficaces les dévermineurs.
- Le *Stealth-bug* [Eisenstadt, 1997], ou *bug fantôme*, qui disparaît après avoir produit son effet et masque ainsi son existence.
- Le *Mandelbug* [Grottke and Trivedi, 2007, Grottke et al., 2008], non-déterministe, qui ne se manifeste que si des conditions spécifiques et parfois complexes de l'environnement sont réunies.

— Le *Schroedinbug* [Raymond and Steele, 1996], décrit comme une erreur manifeste de mise en œuvre qui *"n'aurait jamais dû fonctionner"*. Ces erreurs sont repérées suite à une inspection du code ou à une utilisation inhabituelle du programme, puis *"bloquent le fonctionnement du programme jusqu'à ce qu'elles soient corrigées"*. Raymond et Steele précisent : *"Though [...] this sounds impossible, it happens ; some programs have harbored latent schroedinbugs for years."*.

Leur caractéristique commune est qu'ils effacent toute trace de leur existence après avoir accompli leur méfait. Leur manifestation se produit dans différents contextes. Typiquement, ils passent au travers de toutes les phases de tests [Gray, 1986, Cotroneo et al., 2006, Grottke and Trivedi, 2007], et restent *dormants* dans le programme jusqu'à produire un *scénario surprise* tel que décrit par Knuth [Knuth, 1989]. Ce genre de scénario peut induire en erreur lors du débogage, car si le code du programme a été récemment modifié et qu'un *bug* jusqu'ici inconnu apparaît, la source du *bug* sera logiquement attribuée à la modification alors qu'il était déjà présent à l'état *dormant* avant sa première manifestation. Ce type de *bug* est souvent la cause de l'effet nommé *software aging* [Parnas, 1994, Grottke and Trivedi, 2007, Grottke et al., 2008]. Les systèmes qui s'exécutent pendant de longues périodes accumulent les effets de ces *bugs*, qui finissent par provoquer des fonctionnements erratiques ou des états incohérents. Dans les cas les plus critiques, le programme peut s'arrêter. Des exemples typiques sont les propagations d'erreurs perturbant progressivement le fonctionnement du programme, ou bien des fuites mémoires. Ces *bugs* peuvent être progressivement introduits dans le programme, au gré des maintenances [Parnas, 1994].

L'aspect aléatoire du *Mandelbug*, et parfois du *Heisenbug*, rend également difficile l'observation et la reproduction des problèmes [Grottke and Trivedi, 2007, Grottke et al., 2008]. Il peut de plus exister un délai entre l'apparition d'un symptôme et la manifestation du *bug*. Reproduire ce dernier est alors complexe, car il ne suffit pas de relancer l'exécution du programme et de répéter les mêmes opérations : le *bug* peut ne pas réapparaître entre deux exécutions identiques. Cet aspect aléatoire peut rendre problématique la reproduction d'erreur lors du déverminage [Layman et al., 2013].

Ces *bugs* se manifestent donc de manière non-anticipée, et dans des logiciels en production. Ils peuvent être purement aléatoires, ou intermittents [Agans, 2002], et forcent à expérimenter plusieurs stratégies d'observation et de reproduction pour comprendre leur origine [Agans, 2002, Zeller, 2009]. Parfois même, le *bug* n'apparaît que dans un contexte de production, et jamais en développement. Il n'est alors pas possible de le reproduire *à l'usine*, et il est nécessaire de retrouver les conditions logicielles et matérielles précises dans lesquelles le reproduire. Il n'y

a parfois pas d'autre possibilité que d'instrumenter l'application directement en production *chez le client* [Agans, 2002]. Cette difficulté à reproduire le contexte d'apparition d'un *bug* est source de difficultés dans l'activité de débogage chez les développeur-euse-s [Layman et al., 2013, Perscheid et al., 2017].

Indépendamment de cette classification, certains *bugs* ne sont jamais corrigés bien que leur origine soit parfaitement identifiée. D'autres *bugs* voient leurs symptômes corrigés mais leur origine n'est jamais comprise [Perscheid et al., 2017]. La phase d'observation et de reproduction des *bugs* est donc une étape clé dans le processus de déverminage. Elle est parfois complexe et sujette à de nombreuses difficultés.

1.1.3 Outillage du déverminage

L'outillage est un élément crucial du déverminage pour les développeur-euse-s [Murphy et al., 2006, Layman et al., 2013, Perscheid et al., 2017]. D'une part, l'outillage est intensivement utilisé pour observer et corriger les programmes en erreur, et d'autre part il y a un besoin d'outiller les environnements de développement pour résoudre les problèmes difficiles. Certains *bugs*, tels ceux décrit à la section 1.1.2, sont parfois difficiles à observer car l'utilisation d'un outil d'observation perturbe le système et empêche leur apparition [Gray, 1986, Eisenstadt, 1997, Cotroneo et al., 2006]. En outre, les outils actuels sont parfois difficiles à prendre en main et à maîtriser, et nécessitent un temps d'apprentissage conséquent [Layman et al., 2013, Beller et al., 2018].

1.2 Méthodes, techniques et outils pour le déverminage

Cette section dresse un panorama des techniques et outils existants, et énumère leurs limites (fonctionnelles ou pratiques) pour résoudre les *bugs* les plus difficiles. Bien que la plupart de ces techniques soient agnostiques en terme de paradigme de programmation, nous les discutons sous une perspective orientée objet.

1.2.1 Techniques de base

Les déverminеurs, le *printf* et le *core-dump* sont les premiers outils utilisés lors de l'activité de débogage [Agans, 2002, Zeller, 2009, Layman et al., 2013, Böhme et al., 2017, Perscheid et al., 2017, Beller et al., 2018]. Un-e développeur-euse aura d'abord recours à ces outils pour commencer le déverminage, avant d'utiliser des techniques plus avancées pour les problèmes les plus difficiles.

1.2.1.1 Les dévermineurs

Un dévermineur est un outil qui permet d'interagir avec un programme préalablement configuré pour être débogué, par divers moyens à la disposition de l'utilisateur-rice. La plupart des environnements de développement, depuis Interlisp-D [Teitelman, 1978, Teitelman and Masinter, 1981] et Smalltalk-80 [Goldberg, 1984], fournissent aujourd'hui un outil standard pour la correction de *bug*, nommé dévermineur, débogueur, ou encore *debugger* [Rosenberg, 1996]. Les dévermineurs *open-source* comme commerciaux proposent globalement les mêmes fonctionnalités. Par exemple, les points d'arrêt permettent d'interrompre l'exécution à un point donné du flot de contrôle. Il est alors possible d'observer les données du contexte courant et d'exécuter des instructions. L'exécution pas-à-pas permet, à partir d'un point d'arrêt, d'exécuter l'instruction suivante du flot de contrôle et d'observer l'évolution du programme. Ces fonctionnalités sont standardisées, et se retrouvent parmi les dévermineurs grand public, tels que *GDB* [Stallman and Pesch, 2000] ou le débogueur de l'environnement Eclipse [Eclipse, 2018c].

Points d'arrêt. Les mêmes types de points d'arrêt se retrouvent dans la plupart des outils, à l'instar de ceux présents dans le débogueur Eclipse. Notamment, le *Watchpoint* permet de surveiller les accès en lecture et écriture à une variable, de s'arrêter sur un accès particulier, ou quand la valeur surveillée change. Le point d'arrêt conditionnel (*Conditional Breakpoint*) permet de spécifier une condition particulière, qui est évaluée lorsque l'exécution du programme atteint le point d'arrêt dans le flot de contrôle. Le résultat de l'évaluation détermine si le dévermineur doit arrêter le programme et ouvrir une fenêtre d'interaction à l'attention de l'utilisateur-rice. Le *Hit Count* permet de s'arrêter après être passé *n fois* au travers du point d'arrêt, par exemple pour s'arrêter après un nombre d'itérations spécifique dans une boucle. Xcode [Apple, 2018] et Eclipse [Eclipse, 2018c] permettent de s'arrêter à l'entrée ou à la sortie d'une méthode. Visual Studio [Microsoft, 2018a] propose des filtres pour restreindre les points d'arrêt à un processus ou à une machine spécifique. Visual Studio et Eclipse fournissent le *TracePoint*, un point d'arrêt particulier dont le but est d'imprimer une trace définie par l'utilisateur-rice. Le *TracePoint* instrumente le programme avec des traces non-intrusives et sans stopper le programme, ce qui permet d'observer certains des *bugs* qui disparaîtraient si le programme était interrompu. Certains éléments du contexte sont disponibles pour élaborer la trace, comme l'instruction en cours d'exécution, le nom de la fonction courante, etc.

Dévermineurs extensibles. Certains dévermineurs peuvent être étendus par l'utilisateur-rice, comme ceux du langage Python [PDB, 2018] et de Pharo Smalltalk [Black et al., 2010, Chiş et al., 2015]. L'extension du dévermineur permet d'obtenir des capacités et des visualisations spécifiques au domaine, et ainsi de faciliter le processus de correction.

Dévermineurs à distance. Il existe également des débogueurs distants pour corriger des programmes embarqués, comme par exemple VisualMicro [VisualMicro, 2018] pour *Arduino*[1], Visual Studio [Microsoft, 2018b] ou TelePharo [TelePharo, 2018, Papoulias et al., 2015] pour Pharo Smalltalk [Black et al., 2010]. Ces derniers permettent aux développeur-euse-s d'investiguer un problème à distance sur un programme qui s'exécute dans son environnement de production, ou par exemple en phase de prototypage. Il s'agit d'une possibilité précieuse pour l'observation et la reproduction de *bugs* difficiles, où l'accès au contexte d'un programme en cours d'exécution est primordial pour pouvoir investiguer (section 1.1.2).

Limitations. La première limitation est la nécessité d'embarquer le dévermineur directement dans le programme en cours d'exécution. En général, on parle de charger ou de préparer le programme avec un environnement de débogage [Agans, 2002, Zeller, 2009]. Cette option peut ne pas être une possibilité, par exemple si le programme à investiguer se trouve déjà en environnement de production et que le problème ne se passe que dans des conditions environnementales précises. C'est le cas également si le programme ne peut pas être redémarré sans risques, par exemple un satellite ou une application devant fournir un service de manière ininterrompue.

Il arrive que les développeur-euse-s manquent un point d'intérêt lorsque l'exécution atteint le point d'arrêt dans le flot de contrôle [Beller et al., 2018], ce qui force à changer la position du point d'arrêt puis à relancer l'exécution du programme. Il arrive également qu'un point d'arrêt stoppe l'exécution du programme de nombreuses fois à un moment non pertinent, où le problème n'est pas observable − même avec des points d'arrêt conditionnels [Chern and De Volder, 2007]. Il faut alors *jouer* avec les points d'arrêt pour interrompre le programme une première fois et chercher à en placer d'autres qui, avec de l'espoir, arrêteront le programme à un moment présentant un quelconque intérêt. Ces problèmes et leurs solutions empiriques perturbent et ralentissent l'investigation de *bugs* difficiles, en particulier ceux qui effacent leurs traces.

1. https://www.arduino.cc/

1.2.1.2 Le *printf*, outil manuel universel

La technique communément désignée par *printf* consiste à insérer manuellement des instructions dans le code source du programme, afin de tracer son exécution sous forme textuelle. L'instruction de trace permet d'enregistrer des informations contextuelles du programme exécuté, si l'endroit où les placer est connu. Le choix de placement de traces dans le flot de contrôle dépend donc de l'intuition des développeur-euse-s, et de leur analyse du code. Le *printf* est utilisé comme un moyen simple et universel de commencer l'investigation d'un problème, tandis que le recours à des techniques de déverminage avancées sont moins souvent nécessaires. Quand on les interroge, les développeur-euse-s semblent considérer que les dévermineurs apportent des capacités supérieures à de simples traces, mais continuent néanmoins d'utiliser le *printf* en premier recours [Beller et al., 2018].

Cette technique a deux implications immédiates. Premièrement elle nécessite d'insérer manuellement des instructions directement dans le code source. Cet aspect invasif peut poser problème si le programme doit être instrumenté directement en production pour observer un contexte particulier. Du comportement d'instrumentation peut être oublié dans le code de production, et parasite la lecture du code source lors des opérations de maintenance et d'évolution du programme. Deuxièmement, l'insertion de traces nécessite de recompiler le programme − et donc de l'interrompre − ou la partie du programme qui est modifiée [Zeller, 2009]. L'objectif étant d'isoler progressivement le problème, la démarche implique de modifier, préciser ou rajouter des traces au fur et à mesure des observations. Lors de l'investigation d'un *bug* difficile, la re-compilation du programme doit donc être effectuée chaque fois qu'il faut insérer de nouvelles traces. Cette limitation disparaît dans le cas du *live programming* [Tanimoto, 2013], où le code peut être dynamiquement modifié pendant l'exécution du programme. C'est le cas par exemple du langage Smalltalk [Goldberg and Robson, 1983]. Tous les langages ne sont cependant pas éligibles au *live programming*, que cela soit incompatible avec leur conception ou bien non démontré en pratique.

Enfin, les traces générées pour un programme peuvent être conséquentes, et doivent être interprétées pour en extraire les parties intéressantes. Le problème posé est celui du périmètre des données et évènements tracés, et de la granularité de l'instrumentation. Dans le cas des langages objets, les instrumentations sont placées dans des classes, et sont actives pour toutes les instances de cette classe. L'instrumentation à granularité objet n'est pas possible avec cette technique, alors même que l'investigation d'un problème particulier pourrait le demander.

1.2.1.3 Fichiers de *traces* et *core-dumps*

Un fichier de *traces*, ou *stack-trace* [Eclipse, 2018b], ou *core-dump* [Kerrisk, 2010], est un fichier dans lequel sont sauvegardés de manière textuelle le contexte et l'état du programme au moment de l'arrêt de son fonctionnement. Le contenu de ce fichier peut être rechargé par un débogueur, comme c'est le cas par exemple avec *GDB* [Stallman and Pesch, 2000, Kerrisk, 2010], ou simplement interprété par un-e développeur-euse pour comprendre les évènements et états qui ont mené à l'arrêt du programme.

Cette technique est loin d'être suffisante pour les *bugs* dont les symptômes sont éloignés de leur cause. Le contenu du fichier ne détaille pas tous les états du programme, ni ne reflète tous ses comportements. En particulier, un comportement observé dans le *core-dump* peut être une conséquence d'un comportement erratique dû au *bug*. Un état provoquant un arrêt du programme peut être modifié longtemps avant les derniers évènements inscrits dans le fichier. Pire, un état altéré par un *bug* peut provoquer à terme l'arrêt de l'application, mais être tout à fait *correct* dans les traces s'il a par la suite été modifié par le programme. Ces limitations ne permettent pas d'observer de manière fiable les problèmes difficiles décrits par la section 1.1.2.

1.2.2 Techniques avancées

Pour résoudre les problèmes les plus difficiles, et notamment pour les observer et les reproduire, les méthodes et outils standards ont été étendus et améliorés par de nouvelles techniques. Ces dernières visent toutes à faciliter le processus d'observation des problèmes, en particulier ceux dont la source et les symptômes sont ou apparaissent distants.

Points d'arrêt *avancés* et traces non-intrusives. Ces techniques et méthodes étant universellement connues et utilisées par les développeur-euse-s, elles constituent naturellement une base à augmenter pour faciliter la pratique du déverminage, et notamment en ce qui concerne les problèmes difficiles.

Certains travaux étendent la sémantique des points d'arrêt. Par exemple, les points d'arrêt dits *Control-Flow Breakpoints* [Chern and De Volder, 2007] permettent de conditionner l'activation d'un point d'arrêt à un contexte lié au flot de contrôle du programme en cours d'exécution. Il est alors possible de spécifier un point d'arrêt qui doit s'activer avant un point précis du flot de contrôle, ou bien uniquement après qu'un certain point du flot de contrôle ne soit atteint, par exemple un passage dans une méthode particulière. Cela permet de caractériser un contexte d'exécution pour lequel une partie spécifique du programme doit être observée. Les *Stateful-Breakpoints* [Bodden, 2011], ou points d'arrêt à états, proposent une abstraction supplémentaire qui définit un ensemble de points d'arrêt nommés,

chacun pouvant référer à une variable libre qui stocke un résultat de son contexte d'exécution ainsi qu'une condition. La condition permet d'exprimer, par exemple, l'ordre dans lesquels les points d'arrêt qui composent le *stateful-breakpoint* doivent être atteints pour constituer un comportement fautif. Si l'ordre est respecté, alors une erreur est soulevée et le dernier point d'arrêt atteint ouvre le débogueur.

D'autres techniques permettent d'insérer des traces de manière non-intrusive dans le code. Les *Tracepoint*, disponibles dans les dévermineurs de Visual Studio et Eclipse [Microsoft, 2018a, Eclipse, 2018c], sont des points d'arrêt permettant de définir des traces sans modifier le code source du programme. Tous les éléments du contexte ne sont en revanche pas disponibles pour être utilisés dans l'instruction de trace, et dépendent du débogueur. La programmation orientée aspects [Kiczales et al., 1997] permet d'exprimer des considérations orthogonales au code métier, typiquement des traces [Kiczales et al., 2001a]. Il est possible de factoriser le comportement de traces dans un code à part, qui est inséré automatiquement à des points clés du code source du programme. Les mises en œuvre modernes, telle celle de l'environnement Eclipse [Eclipse, 2018a], permettent à l'utilisateur-rice de spécifier et de visualiser facilement du comportement de trace écrit sous la forme d'*aspects*.

Approches automatiques Les techniques de localisation de faute automatiques [Wong et al., 2016] prennent en entrée un programme dit *en échec*, c'est-à-dire arrêté suite à une erreur ou ne fonctionnant pas comme attendu, et tentent de détecter automatiquement les sources potentielles de l'échec. Une comparaison est éventuellement effectuée avec un programme qui n'est pas *en échec* mais utilisant de manière déterministe les mêmes données d'entrée. Ces techniques ne sont pas applicables s'il n'est pas possible de récupérer des configurations d'entrée, ou s'il n'est pas possible d'obtenir une exécution *en échec* des programmes débogués.

L'*algorithmic-debugging* est une approche semi-automatique qui cherche à isoler une faute dans un programme qui a rencontré une erreur, en posant à l'utilisateur-rice des questions sur la validité des sorties produites par le programme [Shapiro, 1982, Silva, 2011]. Les questions sont générées automatiquement par un algorithme à partir des réponses données par l'utilisateur-rice. L'algorithme utilise les informations entrées pour isoler le problème dans le code source.

Le *Dynamic-slicing* est une technique qui cherche à isoler la partie d'un programme (une *slice*) directement imputable à une faute [Korel and Laski, 1988, Agrawal and Horgan, 1990, Agrawal et al., 1993]. Les *slices* sont calculées à partir de l'analyse de l'exécution du programme. Les instructions directement responsables de l'erreur peuvent alors être isolées et présentées à l'utilisateur-rice.

Le *Delta-debugging* compare deux versions d'un même programme, dont l'une des exécutions *échoue*, afin d'isoler un ensemble minimum de condi-

tions qui provoquent une erreur [Zeller, 1999, Zeller and Hildebrandt, 2002, Cleve and Zeller, 2005, Misherghi and Su, 2006]. La technique permet soit d'isoler un cas d'utilisation minimal reproduisant une erreur, soit d'isoler un ensemble minimal de modifications apportées au programme entre les deux versions comparées et qui produisent une erreur.

Dévermineurs *back-in-time*. Les dévermineurs *back-in-time* permettent de revenir en arrière dans l'exécution d'un programme pour observer plus facilement les causes d'un problème. Il en existe plusieurs formes, dépendantes des modèles et des choix de mise en œuvre.

Les débogueurs dits *reverse/replay* [Feldman and Brown, 1988, Boothe, 2000, Gottbrath, 2008] enregistrent des points clés dans le programme, auxquels il est possible de revenir et de reprendre l'exécution. Il est possible d'avancer *à reculons* au travers de ces points clés (par exemple des points d'arrêt), et de visualiser l'état précédent du programme. Un exemple très récent, les *Transition Watchpoint* [Arya et al., 2017], surveillent les accès à une variable (comme un *Watchpoint* traditionnel) et peut ramener l'exécution du programme en arrière, aux moments clés où la valeur de cette variable a changé.

Les débogueurs dits *record/replay* [Ronsse and De Bosschere, 1999, Wang et al., 2014], enregistrent tout ou partie de l'exécution et peuvent en *rejouer* certaines parties.

Les débogueurs dits *omniscients* [Lewis, 2003, Hofer et al., 2006] enregistrent la totalité de l'exécution sur disque, et permettent de la visualiser et d'y naviguer après l'arrêt du programme. Ces dernières solutions sont très gourmandes en temps d'exécution et en consommation mémoire. De nombreux travaux ont contribué à cette technique. Par exemple, le débogage centré sur les flots d'objets [Lienhard et al., 2006, Lienhard et al., 2008, Lienhard et al., 2009], pour traquer l'origine et la circulation des objets dans une exécution particulière d'un programme. Ce travail vise particulièrement à expliquer comment un objet est arrivé à un endroit spécifique dans l'exécution du programme, et à expliquer l'historique de son état. Un effort est fait pour rendre l'outil pratique à utiliser, dans les visualisations *post-mortem* du flot d'objets et au niveau de ses performances en terme de vitesse d'exécution. D'autres travaux se focalisent sur le passage à l'échelle et le périmètre des informations générées par une exécution [Pothier et al., 2007, Pothier and Tanter, 2009]. Toute l'exécution est sauvegardée dans des bases de données à haute performance, permettant ainsi de passer l'échelle en terme de volume d'information enregistrée. Le débogueur peut également se déplacer en avant et en arrière dans l'exécution enregistrée du programme. Il est possible de définir le périmètre des parties enregistrées du programme, en spécifiant par exemple quelles classes doivent enregistrer leur exécution, ou en activant

manuellement l'enregistrement avec l'interface graphique.

Enfin, des approches expérimentales très récentes, dites *online back-in-time*, fournissent un historique local à une méthode [Schulz, 2017, Schulz and Bockisch, 2017]. Les derniers résultats des évaluations locales sont intégrés dans le dévermineur natif de l'environnement, et disparaissent avec le contexte de la méthode quand cette dernière a fini de s'exécuter. L'historique local permet de visualiser les valeurs ou objets provoquant des *bugs* et qui sont effacés ou modifiés par la suite dans le corps de la méthode (par exemple dans une boucle). Cet historique n'est disponible que si le dévermineur s'arrête, par exemple sur un point d'arrêt ou suivant une exception, et il n'est pas possible de rejouer l'historique.

L'apport de ces solutions, indépendamment de leur type et du choix d'implémentation, réside dans la fonctionnalité de navigation en arrière dans le temps, *post-mortem* ou pendant une (ré-)exécution du programme. La même exécution peut être rejouée ou visualisée plusieurs fois pour comprendre l'origine des problèmes les plus difficiles. Cela signifie cependant que le programme doit s'arrêter pour naviguer dans l'exécution enregistrée, ce qui n'est pas une option pour les systèmes devant fonctionner en continu. Un environnement de débogage particulier doit également être chargé pour enregistrer l'exécution du programme, ce qui peut ne pas être possible dans un système en production. La question de l'enregistrement de l'exécution se pose également pour les programmes qui s'exécutent pendant de longues périodes, par exemple pour observer des problèmes qui n'apparaîtraient qu'au bout de plusieurs jours.

Interrogative-Debugging. *WhyLine* [Ko and Myers, 2004, Ko and Myers, 2008] est un dévermineur qui enregistre l'exécution d'un programme et permet de naviguer dans cet enregistrement après la fin du programme. L'utilisateur-rice peut alors poser des questions *why*? et *why not*? sur l'état du programme, dont la réponse est calculée automatiquement par une analyse statique et dynamique. Par exemple, à la question *"pourquoi telle variable x est dans tel état y ?"*, $WhyLine$ répondra par une visualisation directe du moment de l'exécution où l'état y a été stocké dans la variable x. L'utilisation de $WhyLine$ a permis à des utilisateurs novices de résoudre un *bug* deux fois plus rapidement que des utilisateurs experts sans $WhyLine$ [Ko and Myers, 2008]. L'outil n'est en revanche pas utilisable sur des exécutions durant plus de quelques minutes, et nécessite toujours d'arrêter le programme pour pouvoir analyser son exécution.

Live Debugging. Le *Live Programming* [Tanimoto, 2013] permet de développer et de corriger un programme pendant son exécution. L'impact d'une modification est directement reflété par l'exécution du programme, et permet d'affiner la modification jusqu'à ce que le programme soit correct. Les activités de développement

et de déverminage sont alors indissociables, et permettent l'utilisation d'un déverminateur et de toutes ses capacités (points d'arrêt, exécution pas-à-pas...) avec un accès direct au programme en cours d'exécution. C'est par exemple le cas du langage Pharo [Black et al., 2010] et de son déverminateur. Des travaux portent spécifiquement sur l'utilisation du plein potentiel du *Live Programming* pour le déverminage, en fournissant des interactions et des visualisations dynamiques qui font par exemple le lien entre des traces dynamiques et la portion de code qui les a émises [McDirmid, 2007, McDirmid, 2013]. Mais l'avantage du *Live Programming* est aussi sa faiblesse : la possibilité d'introduire des instrumentations ou des modifications incomplètes dans un code en exécution peut provoquer un arrêt du programme. C'est le cas lorsque des modifications complexes doivent être appliquées à un programme en cours d'exécution, et où la modification d'une partie du code implique au même moment la modification d'une autre partie du code pour fonctionner correctement. *Edit Transactions* [Mattis et al., 2017] fournit un environnement permettant d'encapsuler les modifications à apporter au système dans un *change-set*. Lorsque le *change-set* constitue un ensemble cohérent de modifications, la transaction peut être activée ou désactivée manuellement.

Déverminage d'application *big-data* distribuées. Des travaux récents abordent le débogage sous l'angle du *big-data*[2] et des applications distribuées ne devant jamais être stoppées. Ces architectures se présentent sous la forme d'un réseau de nœuds, un *maître* et un ensemble de *travailleurs*, ainsi qu'un programme en cours d'exécution sur les nœuds du réseau. Les solutions développées permettent d'inspecter et d'interagir avec le programme en cours d'exécution sans stopper ce dernier. Les correctifs peuvent être établis localement sur un nœud dit *client*, c'est-à-dire le poste de développement de l'utilisateur-rice, et déployés automatiquement sur tous les nœuds distants. *BigDebug* [Gulzar et al., 2016a, Gulzar et al., 2016b, Gulzar et al., 2017] est un déverminateur *big-data* pour *ApacheSpark*[3]. L'outil permet de mettre en place des points d'arrêt simulés sur des nœuds distants. Lorsqu'il est atteint, le point d'arrêt simulé transfère l'état du programme provenant de son contexte d'exécution au nœud demandeur, et l'utilisateur-rice peut inspecter et interagir avec ce contexte. Des *Watchpoints* avec des gardes peuvent être mis en place sur les nœuds distants pour récupérer des états intermédiaires du programme. Les gardes permettent de raffiner dynamiquement le périmètre de l'état du programme demandé. *IDRA* [Marra, 2017, Marra et al., 2017, Marra, 2018, Marra et al., 2018] est un déverminateur *out-of-place*, ou *hors application*, pour Pharo Smalltalk [Black et al., 2010]. *IDRA* intercepte les exceptions non bloquantes sur les nœuds distants et les transfère au nœud client avec leur contexte d'exécution.

2. https://fr.wikipedia.org/wiki/Big_data
3. https://spark.apache.org/

Le client les accumule dans une pile, et permet à l'utilisateur-rice de les déboguer localement une par une. Le débogage d'une exception déclenche l'ouverture du déverminer natif de Pharo, dont toutes les fonctionnalités peuvent être utilisées (pas à pas, inspection, etc.) comme si l'exception était soulevée localement. Une limitation immédiate concerne les données d'entrée non-déterministes, comme une lecture sur un capteur. *IDRA* ne permet pas au déverminer local d'exécuter une méthode effectuant une telle lecture indéterministe car le capteur se trouve physiquement sur un nœud distant. Même si une nouvelle lecture physique pouvait être effectuée à distance, étant non-déterministe par nature elle ne permettra pas de reproduire une donnée problématique produisant, par exemple, un *Mandelbug* (section 1.1.2).

Déverminage centré sur les objets. Dans le paradigme objet, plusieurs approches ciblent spécifiquement les objets dans le processus de débogage.

Le *Query-based Debugging* [Lencevicius et al., 1997] permet d'effectuer des requêtes sur le programme en cours d'exécution, qui permettent soit de vérifier des contraintes sur des ensembles d'objets, soit de récupérer un ensemble d'objets satisfaisant les conditions exprimées par la requête. La technique est étendue par le *Dynamic Query-based Debugging* [Lencevicius et al., 1999, Lencevicius, 2000, Lencevicius et al., 2003], qui évalue continuellement les requêtes et met à jour leur résultat. Un débogueur est ouvert dès qu'un résultat retourné par une requête a changé. Les requêtes peuvent être définies avant ou pendant l'exécution du programme, il est donc possible d'affiner les critères de recherche d'objets ou de vérification de contraintes sur ces objets et leurs relations.

L'*Object-Centric Debugging* [Ressia et al., 2012] propose de considérer le débogage du point de vue exclusif de l'objet. Il est considéré avec cette technique qu'un modèle mental de débogage centré sur les objets facilite l'investigation de problèmes difficiles dans le paradigme orienté objet. Un exemple simple montre que l'*Object-Centric Debugging* permet de s'arrêter de manière immédiate sur un accès en lecture à une variable précise dans un objet unique et identifié, alors que 30 opérations d'interactions sont nécessaires pour arriver au même résultat avec un déverminer traditionnel [Ressia et al., 2012]. La solution permet de placer des points d'arrêt dynamiques sur un objet spécifique, par exemple s'arrêtant sur un appel de méthode de l'objet, sur un accès à une de ses variables, etc. et sans affecter les autres instances de la même classe. Ces opérations se font de manière dynamique, pendant l'exécution du programme, et maintiennent toute compatibilité avec l'outillage du langage de mise en œuvre. Notamment, le déverminer natif de l'environnement s'ouvre lorsqu'un point d'arrêt *centré objet* s'active, et la solution ne nécessite pas d'outil dédié. Les objets doivent cependant être trouvés manuellement au cours d'une session de déverminage, à l'aide de points d'arrêt classiques et d'inspection du programme lorsque le premier débogueur s'ouvre. Il n'existe pas, à notre connaissance,

de travaux cherchant à utiliser le *Dynamic Query-based Debugging* pour obtenir dynamiquement des objets sur lesquels placer des points d'arrêt *centrés objet*.

Software Rejuvenation. Ces techniques consistent à *"rajeunir"* un programme en l'arrêtant complètement et en le redémarrant [Cotroneo et al., 2014]. L'état du programme est alors remis à zéro, annulant de fait toute accumulation d'erreurs ou de comportements erratiques dus à un fonctionnement prolongé. C'est un traitement classique de l'effet de *Software Aging* [Parnas, 1994, Grottke and Trivedi, 2007, Grottke et al., 2008] (voir section 1.1.2). Différentes techniques et stratégies peuvent être mises en œuvre suivant le type de programme et son environnement. Cela peut être par exemple un simple redémarrage d'une application ou d'une de ses parties, un redémarrage complet du système d'exploitation incluant un redémarrage du matériel, un redémarrage d'un serveur, remplacé par une copie le temps de la réinitialisation, etc. [Cotroneo et al., 2014].

1.3 Difficultés spécifiques du déverminage en cours d'exécution

Dans cette section, nous décrivons une situation pour laquelle les solutions de l'état de la pratique du déverminage ne sont pas suffisantes. Il s'agit du besoin de déboguer des objets dans les programmes en cours d'exécution. Nous inscrivons alors les propositions de la thèse dans cette motivation. Ces propositions sont restreintes aux programmes mono-processus écrits avec des langages dynamiquement typés.

1.3.1 Déverminage de programmes en cours d'exécution

Nous nous intéressons au problème des programmes en cours d'exécution dans un contexte de production, et qui ne doivent pas être interrompus. Ces programmes rencontrent, en particulier, des problèmes [4] difficiles à observer et à reproduire comme les *bugs fantômes* ou les *Mandelbugs* (section 1.1.2). Nous nous plaçons dans l'hypothèse d'un programme écrit avec un langage objet, et soumis aux contraintes suivantes :

— Le programme rencontre des *bugs*, intermittents ou de manière non-déterministe, dans un environnement de production imprévisible.
— Le programme est déployé dans un environnement de production, possiblement sur un système distant qui n'est pas accessible aux développeur-euse-s (objet connecté, serveur, drone...).

4. Nous ne considérons pas ici les problèmes provoquant l'arrêt définitif de l'application et mettant fin au fonctionnement du programme

— L'accès au contexte de production est primordial pour observer et reproduire le problème, qui est impossible à reproduire *à l'usine*.
— Le programme ne doit pas être interrompu, et il est plus *coûteux* de le laisser se comporter de manière imparfaite à cause d'un problème que de l'arrêter pour investiguer (simulations très longues, services fournis en continu, etc.).
— Si on suppose qu'il est possible de mettre à jour l'application sans la redémarrer, cette dernière n'embarque pas d'environnement de déverminage. Les *bugs* étant par nature non-anticipés, il n'est pas prévu de fonctionnalité de débogage dans le programme.

Une partie des solutions décrites précédemment sont de fait inapplicables dans le cadre de cette hypothèse. Il s'agit des approches qui nécessitent d'interrompre le fonctionnement du programme, soit pour en analyser l'exécution, soit pour utiliser l'environnement de débogage (point d'arrêt, fonctionnalités de retour en arrière dans l'exécution, etc.). Certaines solutions présentent des aspects intéressants pour aborder le déverminage de programmes sous ces contraintes, mais sont limitées par des problèmes orthogonaux. Plusieurs de ces approches nécessitent une certaine anticipation, et sont donc impossibles à mettre en pratique si le programme est démarré et que leur application n'a pas été prévue (*Record/Replay*, *Omniscient-Debugging*...). D'autres approches sont intrusives, et modifient directement le code source du programme (*Live Debugging*). D'autres encore sont agnostiques au regard du paradigme du langage utilisé. Il n'est pas possible de restreindre les opérations de déverminage à un objet particulier, et il est alors difficile de définir le périmètre précis de ces opérations − de surcroît dynamiquement, alors que le programme est déjà en cours d'exécution.

Pour aborder ces contraintes, il est nécessaire de pouvoir appliquer des techniques de déverminage de manière non-anticipée sur le programme en cours d'exécution, au moment opportun et sur les entités adéquates (ici des objets).

1.3.2 Déverminage centré objet

L'*Object-Centric debugging* [Ressia et al., 2012] propose de placer l'objet comme abstraction et entité principale manipulée lors de l'activité de déverminage. Traditionnellement, l'abstraction principale avec laquelle interagir se limite au code source du programme, alors que pendant l'exécution ce code représente des abstractions de plus haut niveau − dans notre cas des objets. Nous souhaitons donc pouvoir considérer, par exemple, qu'une variable dans le code source est un objet, et pouvoir interagir avec ce dernier. L'utilisation des mêmes abstractions avec lesquelles est construit le programme pour instrumenter et déboguer ce dernier évite aux développeur-euse-s de changer de perspective et de modèle mental lors de l'investigation de problèmes difficiles [Yin et al., 2011].

Mais pour manipuler des objets de manière non-anticipée pendant l'exécution du programme, il faut pouvoir les trouver. Le *Query-Based debugging* permet d'effectuer des requêtes sur le programme en cours d'exécution, et ces dernières retournent des objets. C'est une solution intéressante, mais qui n'a jamais été mise en relation avec l'*Object-Centric debugging*. De surcroît, il existe des cas triviaux dans lesquels il n'est pas simple d'exprimer quels sont les objets d'intérêt, ni comment les trouver. Certains objets ont une durée de vie extrêmement limitée. Il s'agit typiquement des objets référencés par des variables temporaires, instanciés dans un périmètre local (par exemple une boucle). Les variables d'instance de ces objets temporaires ont également une durée de vie limitée, car liées à l'objet qui les contient. Il est très difficile de trouver ces objets et de leur appliquer des opérations de déverminage. Ils ne sont accessibles que dans un contexte local, qu'il n'est pas toujours possible de caractériser. D'autres objets ne sont jamais référencés par des variables, et sont en pratique *anonymes*. Il s'agit d'objets issus de l'évaluation d'une expression particulière, directement utilisés dans le corps d'une expression englobante pour produire un autre objet. L'objet anonyme disparaît ainsi après avoir été utilisé. Il n'y a aucun moyen automatique de trouver ces objets avec un débogueur, ou sans instrumentation de code complexe. De telles instrumentations impliquent l'insertion de variables temporaires, et de modifier le code à l'origine de la création de ces objets pour en obtenir des références. Il est également possible de placer des points d'arrêt dans les méthodes de ces objets. Mais cela peut générer de nombreux faux positifs si d'autres instances de la même classe existent ailleurs dans le système et utilisent les mêmes méthodes. Cela implique également de connaître avec certitude la classe de l'objet, ce qui n'est pas toujours le cas dans les langages dynamiques où plusieurs évaluations de la même instruction peuvent donner des objets de différents types [Milojkovic et al., 2017]. La complexité de ce problème explose s'il faut soumettre la récupération d'un de ces objets à des conditions précises − d'autant plus qu'il est nécessaire, lors du processus de débogage, d'affiner dynamiquement ces conditions au fur et à mesure que le périmètre du problème est cerné.

1.3.3 Proposition

Pour déverminer des programmes à objets sous les conditions précédemment évoquées, nous proposons de combiner les idées du débogage centré-objet dans une même solution :

1. Spécification et récupération des objets d'intérêt,
2. Instrumentation des objets d'intérêt avec du comportement de débogage.

Spécification et récupération d'objets. Le code source du programme étant l'abstraction principale à laquelle se réfèrent les développeur-euse-s, nous proposons de conserver cette visualisation mais de considérer ses éléments comme des objets accessibles pendant l'exécution du programme. Par exemple, une variable temporaire dans une boucle référence un objet auquel il est possible d'accéder lorsque ce dernier est affecté à cette variable. N'importe quelle expression et ses sous-expressions produisent, lorsqu'elles sont évaluées, un objet, auquel il est possible d'accéder. Nous proposons de spécifier les objets d'intérêt en ciblant manuellement des expressions du programme lors de l'inspection du code. Comme le programme est en cours d'exécution, cette spécification est complètement dynamique. Les objets sont alors récupérés pendant l'exécution, lorsque le programme passe dans le flot de contrôle désigné et exécute les expressions ciblées. Les objets sont ensuite instrumentés automatiquement par le comportement de débogage spécifié par le ou la développeur-euse.

Instrumentation des objets pour le débogage. Le comportement des objets est décrit dans leur classe. Instrumenter une classe modifie le comportement de tous les objets qui en sont instance. Nous proposons d'instrumenter des objets distincts sous forme d'adaptation de comportement. L'adaptation altère le comportement d'origine d'un objet, sans impact sur les instances de la même classe que celle de l'objet. Du comportement spécifique au débogage peut alors instrumenter les objets, pour observer et reproduire des *bugs*, puis expérimenter des corrections. Ce comportement de débogage est mis en œuvre dans des classes, entités naturelles du paradigme objet, en utilisant les outils du langage et de son environnement de développement. L'adaptation du comportement d'un objet se fera alors par la capacité de l'objet à se comporter temporairement comme une instance d'une autre classe – qui implémente le comportement de débogage. L'instrumentation est alors libre, du moment qu'elle est réalisable par la programmation objet. Les développeur-euse-s n'ont pas besoin de changer de paradigme, ni d'utiliser un outillage spécifique [5], et peuvent déboguer les objets du programme qui leurs sont présentés. Pour pouvoir effectuer ces opérations sur un programme en cours d'exécution, nous proposons l'utilisation de l'adaptation non-anticipée de comportement.

L'adaptation non-anticipée de comportement. L'adaptation ou l'évolution dynamique d'une application consiste à modifier le comportement de cette application sans la stopper, recompiler son code et la redémarrer. Elle est dite

5. La possibilité d'instrumenter librement le comportement d'un objet est une ouverture pour la mise en œuvre des techniques de l'état de la pratique du déverminage. La compatibilité avec l'outillage n'empêche donc pas l'utilisation de la solution proposée en tant que brique de base pour le développement d'outils de débogage spécifiques.

non-anticipée lorsqu'aucune préparation n'est effectuée préalablement au démarrage du programme [Kniesel, 1999, Redmond and Cahill, 2002, Kniesel et al., 2002, Oriol, 2004]. Cette adaptation non-anticipée sera appliquée avec une granularité objet, et permettra d'adapter le comportement d'objets spécifiques dans le programme en cours d'exécution.

Propriétés désirables. Adapter de manière non-anticipée pour déboguer pose certaines contraintes. Le programme à déboguer étant déjà écrit et en cours d'exécution, il n'est pas raisonnable d'attendre que le programme se plie à des contraintes spécifiques concernant son fonctionnement ou sa structure. Il faut également prendre en compte la spécificité de l'activité de déverminage, notamment l'importance de pouvoir modifier fréquemment les instrumentations afin de raffiner le périmètre des hypothèses étudiées. Nous proposons et décrivons des propriétés désirables pour la solution proposée, que nous désignerons par la suite par le terme *adaptation* :

1. **Granularité objet.** L'adaptation peut s'appliquer sur une instance précise, indépendamment des autres instances de la même classe.

2. **Minage d'objets.** Les objets peuvent être récupérés de manière non-anticipée pendant l'exécution du programme, à partir d'entités ciblées dans le code source du programme (par exemple une variable, une expression...).

3. **Réversibilité.** L'adaptation d'un objet peut être *renversée*, et l'objet peut toujours retrouver son comportement d'origine.

4. **Préservation de l'identité des objets.** Les objets adaptés conservent leur identité, notamment leurs relations avec les autres objets du programme, et ne sont pas remplacés par des intermédiaires. La variable spéciale *self*, c'est-à-dire la référence au receveur courant, ainsi que l'interface native déclarée par la classe de l'objet, sont préservées pendant et après l'adaptation.

5. **Flexibilité.** L'utilisation de la solution ne pose pas de contraintes sur la conception originelle du logiciel. Notamment, la possibilité d'adapter n'oblige pas à intégrer du comportement non-fonctionnel ou supplémentaire dans le code de base du programme, par exemple en utilisant une infrastructure ou des instructions supplémentaires.

6. **Non-intrusivité.** L'adaptation d'objets n'introduit pas de code relatif à l'adaptation ni d'instrumentation dans le code de base du programme, et reste

compatible avec l'utilisation des outils du langage et de son environnement (par exemple le déverminer). Il y a séparation entre le code mettant en œuvre l'adaptation de comportement et le code du comportement adapté.

7. **Conservation du paradigme objet.** L'utilisation de la solution n'implique pas de changement de paradigme, et ne force pas le ou la développeur-euse à changer de modèle mental pendant le déverminage.

Originalité de la solution. Dans l'état actuel de nos connaissances, il n'existe pas de solution combinant l'adaptation non-anticipée et le déverminage centré objet pour l'appliquer au débogage d'applications en cours d'exécution. De nombreux travaux et techniques existent cependant concernant l'adaptation non-anticipée. Dans le chapitre suivant, nous discutons de l'état de l'art de l'adaptation de comportement au regard des propriétés désirables que nous avons énoncées.

Chapitre 2

Adaptation non-anticipée de comportement dans les programmes à objets : un état de l'art

Sommaire

2.1 Mise à jour et reconfiguration de programmes en cours d'exécution . . 31
2.1.1 Mise à jour à chaud 31
2.1.2 *Live Programming* 32
2.1.3 Systèmes à base de composants 34
2.1.4 Discussion 37
2.2 Paradigmes permettant la variation comportementale 38
2.2.1 La Programmation Orientée Aspects (AOP) 38
2.2.2 La programmation par les Rôles 41
2.2.3 Programmation Orientée Contexte (COP) 46
2.2.4 Discussion 49
2.3 Approches réflexives 51
2.3.1 Classes anonymes, *wrappers*, *proxies* 52
2.3.2 Protocoles à méta-objets et méta-objets 55
2.3.3 Discussion 61
2.4 Infrastructures dédiées à l'adaptation 62
2.4.1 *Chisel* 62
2.4.2 *LyRT* 63
2.4.3 Discussion. 64
2.5 Synthèse et conclusion 65

L'adaptation de comportement peut être utilisée pour observer et construire des vues d'un système en cours d'exécution, ou lui ajouter du comportement. Les cas d'utilisation les plus courants concernent la compréhension et le déverminage de logiciels − par exemple pour explorer et résoudre des problèmes de performances ou des *bugs* [Redmond and Cahill, 2002, Röthlisberger et al., 2008, Nierstrasz et al., 2009a, Würthinger et al., 2013], pour surveiller l'exécution d'un programme et l'accès à des variables d'instance [Rosà et al., 2016] ou pour fournir des vues à la demande à des utilisateur-rice-s expert-e-s du système [Gjerlufsen et al., 2009]. De telles instrumentations ne sont cependant pas toujours possibles, ni désirables. C'est le cas par exemple si le programme n'a pas été conçu avec de telles capacités, ou si l'instrumentation implique l'altération du comportement de base du programme, ses performances ou sa stabilité. Dans le cas où cette instrumentation est possible, il est également nécessaire qu'elle soit suffisamment flexible pour être mise en œuvre, et pour qu'un-e développeur-euse puisse exprimer les conditions sous lesquelles il ou elle souhaite inspecter et modifier le système. Nous avons donc, au chapitre précédent, explicité sept propriétés désirables pour déverminer un programme en cours d'exécution avec l'adaptation de comportement (section 1.3.3).

Dans ce chapitre, nous faisons un état de l'art de l'adaptation de comportement, que nous évaluons au regard des propriétés recherchées. Nous faisons le choix de restreindre le périmètre de cet état de l'art aux techniques, solutions et mécanismes satisfaisant les contraintes suivantes :

— Les langages et programmes écrits avec ces langages se situent dans le cadre du paradigme objet et de ses extensions,
— l'adaptation peut s'appliquer de manière non-anticipée pendant l'exécution.

Nous faisons donc abstraction des solutions statiques, dont l'aspect anticipé ne permet pas de décider d'une adaptation non prévue si le programme est déjà en cours d'exécution, sauf dans le cas où ces aspects statiques doivent être évoqués pour décrire un domaine ou un paradigme de programmation. Pour chaque propriété, les solutions étudiées sont évaluées selon trois notations. Une propriété peut être totalement satisfaite, si au moins une solution issue de la littérature ou de l'industrie est capable de l'atteindre. Nous considérons alors que, parmi l'ensemble de solutions d'un domaine ou d'une technique, la propriété peut être satisfaite. La propriété peut être partiellement satisfaite, si elle n'est pas totalement atteinte ou bien si elle n'est atteignable que sous certaines contraintes (par exemple, en violant une autre des propriétés recherchées). Enfin, une propriété n'est pas satisfaite si aucune solution ne peut l'achever, ou uniquement sous des contraintes très fortes (par exemple en requérant l'utilisation d'une machine virtuelle dédiée).

Les solutions étudiées sont discutées, lorsque cela est approprié, suivant deux axes orthogonaux : la possibilité d'utiliser une solution comme support pour l'adaptation non-anticipée et la possibilité d'utiliser une solution comme base de mise en œuvre.

2.1 Mise à jour et reconfiguration de programmes en cours d'exécution

Dans cette section, nous décrivons des solutions pouvant modifier le comportement d'un programme en cours d'exécution de manière non-anticipée, soit par une mise à jour directe du code du programme, soit par la reconfiguration du programme. Il s'agit de la mise à jour à chaud, du *Live Programming* et de la programmation orientée composants.

2.1.1 Mise à jour à chaud

La mise à jour à chaud de logiciel, ou *Dynamic Software Update* (DSU), consiste à mettre à jour un programme en cours d'exécution sans avoir à le redémarrer [Miedes and Munoz-Escoı, 2012, Seifzadeh et al., 2013]. Le cycle *arrêt-recompilation-redémarrage* est évité, et le logiciel peut continuer son exécution de manière (quasi-)ininterrompue. Pour les systèmes à objets, la structure du logiciel peut être, selon la souplesse de la solution utilisée, réorganisée et ses classes modifiées ou supprimées. Les objets déjà instanciés d'une classe modifiée sont traqués et migrés vers la nouvelle version de la classe, tout en maintenant leur état. Une fois le programme migré vers une nouvelle version, il est possible (selon la solution utilisée) de revenir en arrière sur tout ou partie des modifications et de revenir à une version antérieure. Le DSU n'est pas spécifique aux langages objet et existe pour différents paradigmes. On trouve des solutions de DSU pour, par exemple, le langage C [Neamtiu et al., 2006], les langages objet dynamiques comme Python [Martinez et al., 2013, Martinez et al., 2015] ou les langages statiquement typés comme Java [Pina and Hicks, 2013, de Pina, 2016]. Des travaux récents s'intéressent à l'efficacité de la technique et à perturber le moins possible le logiciel en cours d'exécution [Gu et al., 2014], ou au sujet particulier de la migration d'instances et des nécessités au niveau langage pour supporter la migration [Tesone et al., 2016]. Certaines approches se présentent sous la forme d'extensions de langages [Pina and Hicks, 2013, Martinez et al., 2013, Martinez et al., 2015, de Pina, 2016], d'autres reposent sur une intégration du DSU au niveau langage par le développement de machines virtuelles spécifiques [Würthinger et al., 2013]. La modification de machines virtuelles rend la fonctionnalité de DSU moins portable, mais permet des optimisations en termes de sécurité et de performances.

Il est possible d'utiliser le DSU pour mettre en œuvre de l'adaptation de comportement non-anticipée. D'une certaine manière, changer dynamiquement le comportement d'un logiciel est une forme de mise à jour à chaud. Cependant cette dernière n'agit que par la migration d'une version du logiciel à une autre − principe fonda-

teur de la mise à jour. Il s'agit d'une modification invasive du code du programme, qui permet certes d'instrumenter ce dernier et de raffiner les instrumentations lors du débogage, mais qui se fait au prix de la modification explicite du code source et de sa lisibilité. Par exemple, la recherche d'objets d'intérêt à déboguer peut s'avérer difficile. Cela peut nécessiter l'insertion de nombreuses instructions conditionnelles à des endroits différents du programme, puis à gérer des variables spécifiques dans lesquelles stocker ces objets. Cela prépare les opérations de déverminage par du comportement non-fonctionnel, complété par un comportement de débogage sur les objets récupérés (par exemple des traces). Lorsque la session de déverminage est terminée, il faut ensuite retirer toutes ces instrumentations par une autre mise à jour, en s'assurant de ne rien oublier. Ce sont des opérations possibles en théorie, mais lourdes à gérer en pratique sans l'outillage adéquat. Enfin, le DSU ne permet pas de modifier radicalement et différemment le comportement d'instances spécifiques de la même classe sans modifier la structure du programme.

Un avantage du DSU pour l'adaptation dynamique est la possibilité immédiate de gérer les différentes versions des mises à jour. Comme les mises à jour sont clairement définies et identifiées, il est possible d'établir un périmètre des modifications devant être à terme définitivement intégrées au logiciel, ou bien ne devant être déployées que sur des installations spécifiques (par exemple en production). Un autre avantage est que les modifications apportées aux classes du programme sont compilées *hors-ligne*, en dehors de l'environnement d'exécution du programme, ce qui économise du temps d'exécution. Si la mise à jour à chaud ne satisfait pas la totalité des propriétés désirables de la section 1.3.3, elle représente un support possible pour mettre en œuvre l'adaptation non-anticipée de comportement. Cette dernière nécessite en effet un moyen de communication avec le programme en cours d'exécution, et d'injecter des variations comportementales − c'est-à-dire, d'une certaine manière, de mettre à jour le code, principe même du DSU.

2.1.2 *Live Programming*

Les environnements de *Live Programming* [Tanimoto, 2013, Rein et al., 2018] permettent de modifier un programme en cours d'exécution. Les activités de développement et de déverminage sont entremêlées. Une modification apportée au code source est immédiatement reflétée sur le programme en cours d'exécution. Smalltalk [Goldberg and Robson, 1983], Self [Ungar and Smith, 1987], *Morphic* [Maloney and Smith, 1995] et Lively [Ingalls et al., 2008, Ingalls et al., 2016] représentent des cas typiques d'environnement de *Live Programming*. De rares solutions commerciales (très récentes) comme *JRebel* [zeroturnaround, 2018], qui bien que mettant surtout en œuvre des techniques de DSU, se rapprochent du *Live Programming* dans l'idée d'un cycle de développement et de déverminage intégré à l'exécution du programme. Proche de notre problématique d'adaptation pour le dé-

bogage, *Edit Transactions* [Mattis et al., 2017] permet de rassembler les variations comportementales en un ensemble de modifications cohérent avant d'être appliqué au programme. Les *transactions* sont isolées et versionnées, et peuvent être dynamiquement raffinées ou étendues. *Lively Groups* [Felgentreff et al., 2015] est une extension pour *Lively*, qui permet de grouper des objets pour leur appliquer une modification comportementale commune. Les objets doivent être marqués par un identifiant correspondant au groupe auquel ils appartiendront. Le marquage est manuel ou programmatique, mais les auteurs ne détaillent pas assez cette possibilité pour pouvoir l'évaluer.

Le *Live Programming* est difficile à envisager comme moyen final pour l'adaptation non-anticipée d'objets. Un environnement de *Live Programming* basé sur un système de classes-objets permettra certes de rester dans le paradigme objet, mais il paraît difficile de sélectionner des objets uniques pour les modifier sans changer leur classe, ou sans modifier la hiérarchie de classes. En outre et à la différence du DSU, les environnements de *Live Programming* ne possèdent pas *nativement* de moyens pour gérer la migration et le versionnage d'objets. Les langages à prototypes [Dony et al., 1998], dont fait partie Self, sont un genre de programmation objet non basée sur des classes. En lieu et place, un langage à prototype se base uniquement sur des objets, qui contiennent des champs. Un champ est basiquement un objet, qui peut représenter un état ou un comportement. Un objet peut être extension d'un autre objet, et en partager des propriétés (des champs). Le problème de l'adaptation à granularité objet se pose différemment qu'avec les systèmes de classes-objets, mais avec un effet similaire. S'il est aisé de modifier le comportement d'un objet en bout de chaîne d'extension, c'est-à-dire qu'aucun autre objet n'en est l'extension, modifier une propriété d'un objet au milieu de la chaîne modifiera également la propriété pour tous les objets qui l'étendent. *Lively* est un environnement visuel de programmation web, dans lequel il est possible de modifier manuellement le comportement d'un objet spécifique. Mais cela repose entièrement sur l'environnement *live*, et le fait que l'utilisateur-rice puisse sélectionner physiquement un objet à l'écran. Cette méthode ne passe pas l'échelle, et son extension *Lively Groups* ne précise pas les moyens à disposition pour la sélection programmatique de groupes d'objets. Dans tous les cas, l'adaptation par le *Live Programming* est invasive par définition, car il s'agit bien de modifier directement le code source du programme. Cependant, et à l'instar du DSU, le *Live Programming* représente un support possible sur lequel un mécanisme d'adaptation pourrait reposer pour changer le comportement d'objets dans un programme en cours d'exécution.

2.1.3 Systèmes à base de composants

Un système logiciel à composants est un programme composé d'unités indépendantes et inter-connectées nommées *composants* [Szyperski et al., 1999, Duncan, 2003, Szyperski, 2003, Lau and Wang, 2005, Lau and Wang, 2007]. Une définition possible, formulée par Szyperski [Szyperski and Pfister, 1997] et rapportée par Duncan [Duncan, 2003], décrit le composant logiciel de la manière suivante :

"A software component is a unit of composition with contractually specified interfaces and explicit context dependencies only. A software component can be deployed independently and is subject to composition by third parties."

Il s'agit d'une *boîte noire*, dont les seuls échanges avec *l'extérieur* se font via des interfaces. Ces interfaces peuvent représenter des services requis par le composant pour fonctionner, c'est-à-dire des dépendances [Szyperski, 2003], et des services fournis. Elles permettent aux composants d'interagir entre eux [Duncan, 2003, Lau and Wang, 2005, Lau and Wang, 2007]. La Figure 2.1 illustre un système à base de composants, suivant un formalisme simplifié. La Figure 2.1.*a)* illustre un composant qui fournit une interface de services et requiert les services d'un autre composant pour fonctionner. La Figure 2.1.*b)* illustre la connexion de deux composants. Chacun des deux composants fournit des services à l'autre composant, et utilise les services fournis par ce dernier.

Un composant repose sur un modèle ; plusieurs modèles et terminologies existent [Lau and Wang, 2005, Lau and Wang, 2007]. Le modèle est traditionnellement constitué de classes, qui mettent en œuvre le comportement et l'état du composant. Ce comportement est isolé du reste du système, et n'est accessible que par des interfaces modélisées par des *ports* [Lau and Wang, 2007]. Les ports permettent aux composants de s'interconnecter, et d'utiliser ou de requérir les services d'autres composants. L'instanciation du composant correspond à l'instanciation de son modèle, c'est-à-dire à la création d'un ensemble d'objets instances des classes qui le composent [Duncan, 2003, Szyperski, 2003]. Ces objets forment un groupe d'instances indépendantes du reste du programme, purement liées à l'existence du composant et n'ont aucun contact avec *l'extérieur* de ce dernier.

Le composant est réutilisable [Szyperski and Pfister, 1997, Duncan, 2003, Lau and Wang, 2007] − au sein du même système ou dans d'autres systèmes logiciels − et *modulaire* [Duncan, 2003, Lau and Wang, 2007], et peut être remplacé par un autre composant.

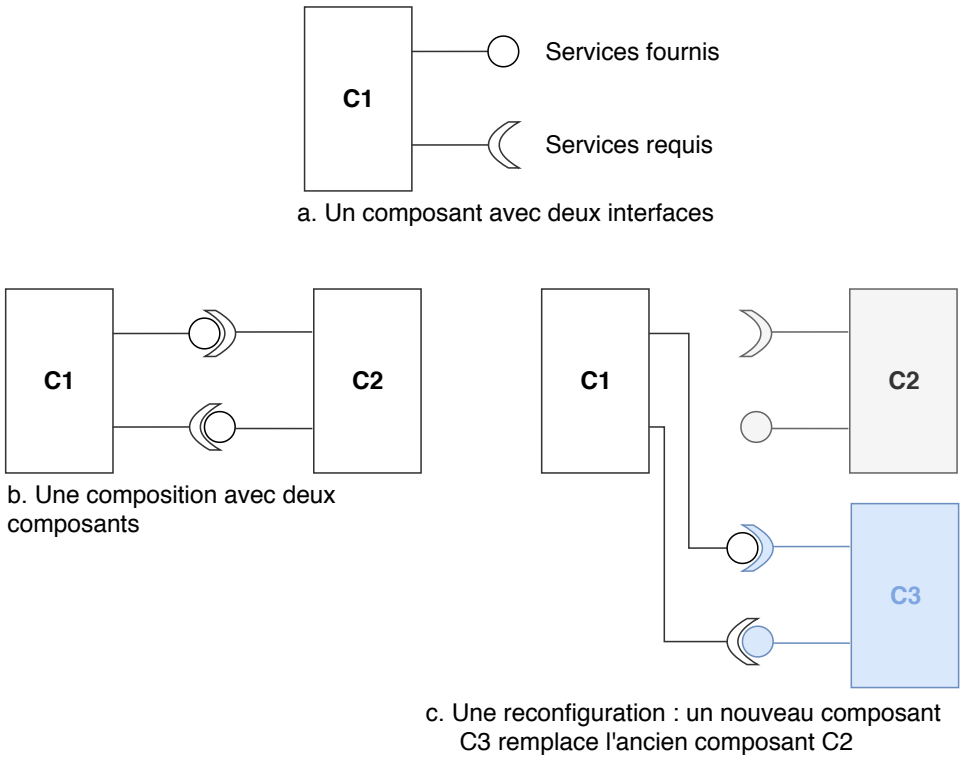


FIGURE 2.1 – Une illustration d'un système à composants. a) Un composant avec une interface de services fournis et de services requis. b) Deux composants connectés, qui requièrent et fournissent des services l'un à l'autre. c) Un nouveau composant (*C3*) remplace un ancien composant (*C2*) lors d'une reconfiguration.

L'adaptation dans les systèmes à composants. L'adaptation dans les systèmes à composants est étudiée de longue date, pour faire évoluer un programme, le corriger ou l'adapter à de nouvelles situations [Szyperski et al., 1999, Szyperski, 2003]. Les modifications apportées aux composants peuvent être fonctionnelles ou structurelles [Vandewoude and Berbers, 2002]. Une adaptation fonctionnelle consiste à ajouter ou modifier du code dans un composant afin d'en modifier la fonctionnalité. Cela peut se faire par exemple en mettant à jour un composant particulier, ou en le remplaçant dans le système comme illustré par la Figure 2.1.c). L'adaptation structurelle consiste à modifier les connexions entre les composants, impactant ainsi la structure du système. Cela reviendrait par exemple, dans la Figure 2.1.c), à modifier les liens et les dépendances entre composants. L'évolution comportementale d'un système à composants est donc atteinte par modification ou reconfiguration de ses composants.

Ces deux types d'évolution ont été étudiées dans le cas particulier de l'adaptation non-anticipée, et différentes préoccupations ont été abordées par ces recherches. L'adaptation fonctionnelle non-anticipée a été abordée sur plusieurs aspects, par exemple le (re)routage dynamique de ports [Vandewoude and Berbers, 2002] ou le remplacement d'un composant [Vandewoude and Berbers, 2004]. Des algorithmes ont été conçus et étudiés pour résoudre des problématiques liées à la sécurité de l'adaptation dynamique et au transfert d'état [Vandewoude and Berbers, 2002, Vandewoude and Berbers, 2004, Vandewoude, 2007]. La reconfiguration a été formellement étudiée du point de vue du transfert d'état dans les systèmes concurrents [Hammer, 2009, Hammer and Knapp, 2010] et de la cohérence de la reconfiguration [Rudametkin Ivey, 2013]. Des recherches ont porté sur le besoin de supporter et de faciliter la modification non-anticipée des connexions entre composants [Fabresse et al., 2006]. D'autres travaux ont mis en œuvre la reconfiguration non-anticipée par une adaptation comportementale des ports des composants [Piechnick et al., 2012]. Cette technique remplace les ports par des *rôles* (voir section 2.2.2) qui peuvent dynamiquement changer pendant l'exécution du programme. L'adaptation comportementale du système est permise par la capacité des ports à jouer des *rôles* différents de manière non-anticipée, ce qui engendre une reconfiguration dynamique des composants pendant l'exécution du programme.

Limites pour l'application au déverminage. Deux limites se posent pour le débogage ou l'étude de solutions et de création d'outils pour le débogage avancé.

Tout d'abord, si la programmation orientée composant fournit des solutions efficaces et sûres pour l'adaptation non-anticipée de comportement, ces dernières ne sont pas applicables aux systèmes qui ne sont pas à base de composants. Les programmes dont la conception est strictement orientée objet ne permettent pas toujours de basculer vers un système à composants. Si une telle migration est possible [Zellagui et al., 2017], il n'y a pas de garantie de pouvoir l'appliquer à tous les programmes. Ce n'est pas non plus raisonnable d'imposer une telle contrainte pour une activité de débogage spécifique. Il y a donc une limite en terme de flexibilité, car la puissance des systèmes à composants n'est disponible que si le programme est déjà à base de composants ou s'il est possible de migrer le programme d'une mise en œuvre à base d'objets vers une mise en œuvre à composants.

Ensuite, la programmation orientée composant peut être considérée comme un paradigme de plus haut niveau que la programmation objet. La granularité, et notamment au niveau de l'adaptation, se fait au niveau du composant ou des connexions entre composants. L'adaptation au niveau objet serait donc plus fine que l'adaptation au niveau composant. Cela est discutable dans la mesure où un composant étant développé à base de classes, qui deviennent des objets lors de son instanciation, il est possible d'imaginer une adaptation par refonte des classes et

mise à jour du composant. Mais c'est désormais le paradigme objet qui pose une limitation, car par définition la modification d'une classe entraîne la modification de toutes ses instances. L'adaptation non-anticipée au niveau objet peut donc être intéressante pour le débogage des objets contenus par un composant.

● Totalement supporté ◑ Partiellement supporté ○ Non supporté

	Granularité objet	Minage d'objets	Réversibilité	Identité préservée	Flexibilité	Non-intrusivité	Paradigme objet
DSU	○	○	●	◑	●	○	●
Live Programming	○	○	●	◑	●	○	●
Prog. orientée composants	○	○	●	◑	◑	◑	○

TABLE 2.1 – Positionnement des approches de type mise à jour à chaud et reconfiguration dynamique au regard de nos contraintes pour l'adaptation non-anticipée.

2.1.4 Discussion

La table 2.1 résume le positionnement des approches de type mise à jour à chaud et reconfiguration dynamique pour l'adaptation non-anticipée de comportement. Nous considérons la propriété du minage d'objets comme non satisfaite pour le *Live Programming* et le DSU, qui peuvent le réaliser en pratique mais au prix de l'insertion et de l'instrumentation invasive du code source. D'autres aspects sont non-satisfaits mais apparaissent néanmoins de manière transparente comme satisfaits à l'utilisateur-rice. C'est le cas de la préservation de l'identité des objets. Un objet ou un composant peut être remplacé pendant l'exécution, mais sa mise à jour et la migration de son état peuvent être totalement transparentes d'un point de vue extérieur, donnant l'impression d'interagir avec la même entité. L'aspect non-intrusif pour les systèmes à composants est également en demi-teinte et dépend de la technique d'adaptation comportementale. Compte tenu de la granularité au niveau composant et de ses aspects *boîte noire*, il est possible de considérer qu'une modification touchant des objets internes à un composant est non-intrusive, car non-visible de l'extérieur au niveau du grain composant.

Sur des aspects orthogonaux concernant la conception de nouvelles solutions de débogage et de leur mise en œuvre, le *Live Programming* et le DSU sont des

candidats pour servir de support à l'adaptation non-anticipée. Il est en effet nécessaire de pouvoir communiquer avec le programme en cours d'exécution pour pouvoir l'adapter. S'il est toutefois possible de mettre en œuvre des solutions dédiées, il faut considérer que le *Live Programming* et le DSU sont des domaines étudiés depuis longtemps et pour lesquels il existe de nombreux résultats exploitables.

2.2 Paradigmes permettant la variation comportementale

Dans cette section nous décrivons des paradigmes qui étendent l'orienté objet, et qui permettent l'évolution ou l'adaptation comportementale à granularité objet. Ces paradigmes sont la programmation par *aspects*, la programmation par les *rôles* et la programmation orientée contexte.

2.2.1 La Programmation Orientée Aspects (AOP)

La programmation par aspects [Kiczales et al., 1997] permet d'intégrer de manière indépendante et modulaire des préoccupations transversales au code métier. Ces préoccupations transversales concernent tout type d'opération répétitive sans lien direct avec le code métier, mais devant être exécutée à plusieurs endroits différents du code. Les traces d'exécution pour le débogage sont typiquement un cas de préoccupation transversale, décorrélé des fonctionnalités du programme mais invasif dans sa mise en œuvre. L'insertion de traces pollue le code métier et rend difficile sa lecture et sa compréhension, et peut être difficile à gérer lors du déploiement d'une application pour lequel il est nécessaire de retirer le comportement de débogage. Un aspect représente donc une préoccupation transversale, dont le comportement est isolé et cloisonné dans un *advice*. L'*advice* est exécuté à un point précis du flot de contrôle nommé *pointcut*, choisi parmi les points d'injection possibles du programme (nommés *joinpoints*). L'injection de comportement est nommée *weaving*, ou *tissage*, et modifie le programme de base avec les préoccupations représentées et mises en œuvre par les aspects. La Figure 2.2, extraite de [Soria, 2011], illustre le principe de l'AOP et de préoccupation transversale du point de vue du flot de contrôle. Les instructions sans rapport fonctionnel avec le code métier (en rouge, à gauche) sont encapsulées dans un aspect (boîte bleue, à droite). Le comportement de l'aspect est exécuté lorsque les *joinpoints* (points rouges) ciblés par des *pointcuts* (flèches rouges) sont atteints dans le flot de contrôle. Cela correspond aux endroits où étaient originellement placées les instructions correspondantes dans le code à gauche.

La Figure 2.3 illustre un aspect mis en œuvre en Java avec AspectJ [Kiczales et al., 2001a, Kiczales et al., 2001b, Eclipse, 2018a]. Nous souhaitons visualiser les objets qui font appel à une méthode *getValue()* dans deux classes

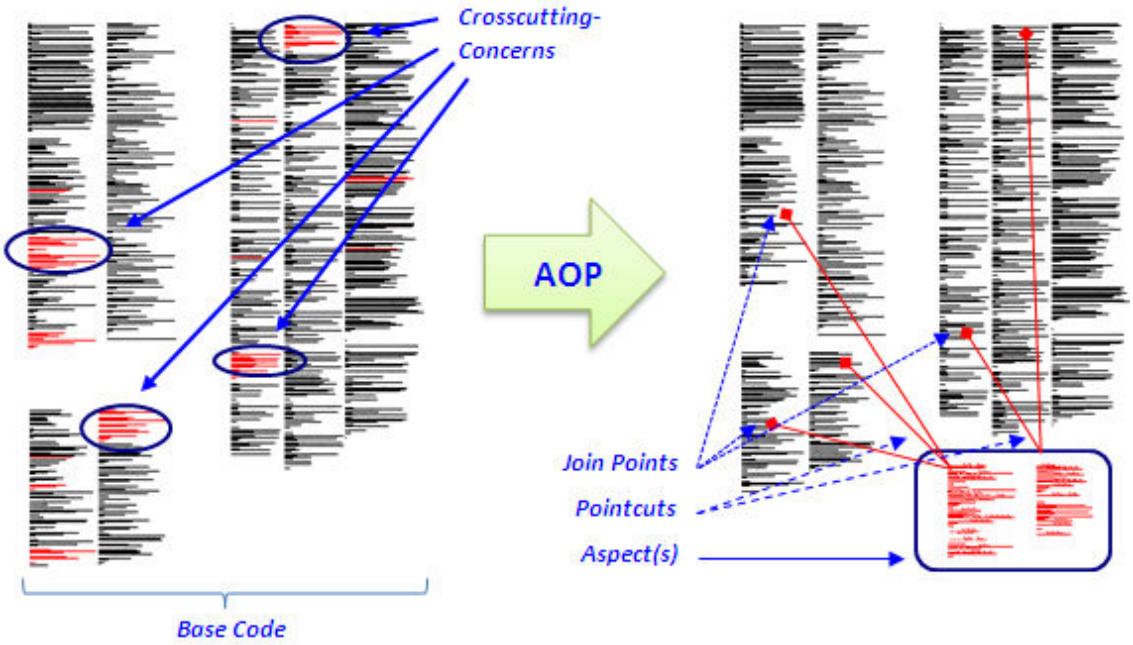


FIGURE 2.2 – Principe de l'AOP, extrait de [Soria, 2011]

du même programme. Plutôt que d'insérer des traces dans le code source, nous créons un aspect (en bas à droite) nommé *LoggingAspect*, qui définit un *pointcut*. Ce *pointcut* cible tous les *joinpoints* correspondant aux objets faisant appel à la méthode *getValue()* (lignes 5-6). Le comportement de l'aspect (ligne 9) accède à la cible réifiée du *joinpoint*, c'est-à-dire l'objet qui a fait appel à la méthode *getValue()* lorsque le *pointcut est atteint*, et l'imprime dans la console.

Aspects dynamiques. Les aspects sont, dans leur définition première, purement statiques. Une fois le programme démarré, il n'est plus possible de leur apporter de modification sans arrêter, recompiler et redémarrer le programme. Les aspects dynamiques [Filman et al., 2004] considèrent le besoin de faire évoluer les préoccupations que les aspects expriment pendant l'exécution du programme. Le tissage ou dé-tissage peut être totalement dynamique [Hirschfeld, 2002, Popovici et al., 2002, Fabry and Galdames, 2014]. Les *pointcuts* et les *advices* peuvent être dynamiquement modifiés, sans que cela soit prévu ou contraint avant le démarrage du programme. D'une autre manière, plutôt que d'injecter dynamiquement les *advices*, AspectWerkz pour Java [Bonér, 2004, Vasseur, 2004], ne *tisse* que des *joinpoints*. Ces derniers sont ciblés par des *pointcuts* statiques, auxquels sont liés des *advices*. Lorsque le *jointpoint* correspondant est atteint à l'exécution, une indirection de *lookup* est effectuée pour retrouver les *advices* qui lui sont liés et exécuter leur

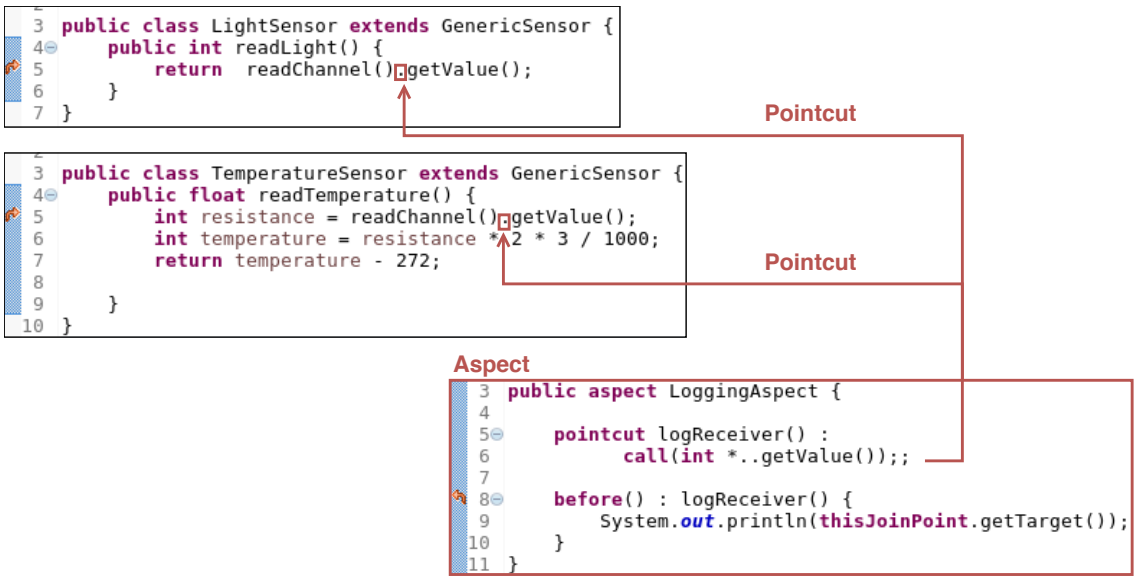


FIGURE 2.3 – Illustration de traces d'exécution mises en œuvre avec un aspect. Le comportement de l'aspect (en bas à droite) s'exécutera chaque fois qu'un *joinpoint* ciblé par le *pointcut* défini dans l'aspect sera atteint dans le flot de contrôle. Ce *pointcut* cible les objets qui appellent la méthode *getValue()*, et le comportement implémenté imprime la référence de ces objets dans la console.

comportement. Les aspects et *advices* peuvent alors être ajoutés, modifiés ou supprimés pendant l'exécution en les liant ou déliant dynamiquement des *joinpoints*. Les *pointcuts*, qui déterminent les *joinpoints* où l'indirection de *lookup* est mise en place, peuvent également être modifiés après le démarrage du programme, par la mise en œuvre de techniques de DSU. Cela permet l'expression des préoccupations transversales de manière totalement dynamique, et ne nécessite plus de les anticiper avant le démarrage du programme.

Des exemples d'utilisation avancée des aspects dynamiques sont, de manière non-exhaustive, la mise en œuvre de solutions d'évolution et de reconfiguration logicielles dynamiques [Costa-Soria et al., 2011, Soria, 2011] ou d'outils de débogage avancés [Yin, 2013].

Les aspects pour l'adaptation non-anticipée de comportement. Les aspects dynamiques permettent l'adaptation de comportement non-anticipée, ou constituent tout du moins une base de mise en œuvre intéressante. Par exemple, les *introductions* d'AspectWerkz [Bonér, 2004, Vasseur, 2004] sont des constructions permettant d'étendre une classe avec de nouvelles méthodes, interfaces ou superclasses, tout en restant dans du code Java standard. L'indirection de *lookup* exécutée

sur les *jointpoints* permet d'ajouter, de modifier ou de remplacer dynamiquement les modifications apportées par les *introductions*. Des variations comportementales peuvent alors être mises en place de manière complètement non-anticipée.

Certaines solutions permettent une application des aspects avec une granularité objet. AspectS [Hirschfeld, 2002] fournit des *Advice Qualifiers*, qui spécifient des propriétés dynamiques pour les *jointpoints* visés par un *pointcut*. Il est alors possible de restreindre le périmètre d'un *advice* à certains objets particuliers. Eos [Rajan and Sullivan, 2003] permet d'instancier des aspects, et de leur associer dynamiquement les objets qui doivent être impactés par les *advices*. Seules les instances associées à l'aspect sont impactées par l'exécution d'un *advice*, c'est-à-dire dans le cas qui nous intéresse par la modification comportementale.

L'adaptation non-anticipée du comportement d'objets est donc possible avec les aspects dynamiques.

Limites pour l'adaptation non-anticipée d'objets. S'il est possible d'utiliser l'AOP pour adapter le comportement d'objets, cette adaptation est exprimée hors du paradigme objet. Premièrement, les variations comportementales sont décrites sous forme d'aspects, et non pas de classes. La Figure 2.3 illustre le code d'un aspect, déclaré dans une construction de code particulière (en bas à droite). Secondement, l'objet n'est jamais la cible première de la modification de comportement. Tout est exprimé du point de vue des aspects : c'est à l'aspect que les objets cibles sont associés (Eos), ou c'est l'*advice* qui est paramétré pour filtrer des objets (AspectS).

S'il est donc possible d'adapter avec une granularité objet, l'AOP ne permet pas nativement de trouver ces objets. Cela pourrait tout de même être mis en œuvre. Par exemple, dans la Figure 2.3, le *pointcut* cible expressément les *joinpoints* qui correspondent à un objet faisant appel à la méthode *getValue()*. Dans les deux classes illustrées, il s'agit d'un objet *anonyme*, c'est-à-dire qu'il n'est stocké dans aucune variable, issu de l'exécution de la méthode *readChannel()* et faisant appel à *getValue()*. Au lieu de tracer l'objet, le comportement de l'aspect pourrait stocker sa référence dans une structure de données pour utilisation ultérieure. Il n'existe cependant pas de modèles qui décrivent comment spécifier quels objets rechercher, ni sous quelles contraintes les récupérer.

2.2.2 La programmation par les Rôles

Les rôles représentent des points de vue sur des objets [Carré, 1989, Bardou and Dony, 1996]. Le concept de rôle a été introduit pour répondre au besoin de placer des perspectives sur des objets, spécifiques à des contextes particuliers [Bäumer et al., 1998, Graversen and Osterbye, 2003]. Un rôle représente donc une vue sur un objet, spécifique à un contexte, et que l'objet peut acquérir ou

perdre pendant l'exécution du programme. L'objet peut donc se comporter différemment suivant le contexte, en gagnant et perdant des rôles dynamiquement tout en conservant son identité [Bäumer et al., 1998, Graversen and Osterbye, 2003].

Les différentes représentations et utilisations des rôles [Fowler, 1997, Bäumer et al., 1998, Steimann, 2000] ainsi que leurs nombreuses propriétés [Steimann, 2000, Kühn et al., 2014] ont été décrites dans la littérature. Steimann résume l'utilisation des rôles à trois grands cas de figure [Steimann, 2000] :

— Le rôle peut être vu comme une relation nommée dans un modèle,
— comme une spécialisation ou une généralisation d'un type dans un modèle,
— ou comme addition à une instance d'un type, qui peut être associée à cette instance pour la compléter.

Dans les deux premiers cas le rôle est utilisé pour décrire et concevoir des modèles, notamment par ses collaborations et ses relations avec les autres rôles [Kendall and Heath, 1998, Tamai, 1999]. Le dernier cas, qui nous intéresse plus particulièrement, consiste à altérer l'état et le comportement des objets pour des contextes particuliers. Les rôles peuvent être instanciés et ces instances peuvent être dynamiquement associées avec un objet (ou dissociées de cet objet) [Fowler, 1997, Bäumer et al., 1998, Steimann, 2000]. Une fois un rôle associé à un objet, l'ensemble apparaît comme un seul objet, parfois appelé *sujet* [Bäumer et al., 1998].

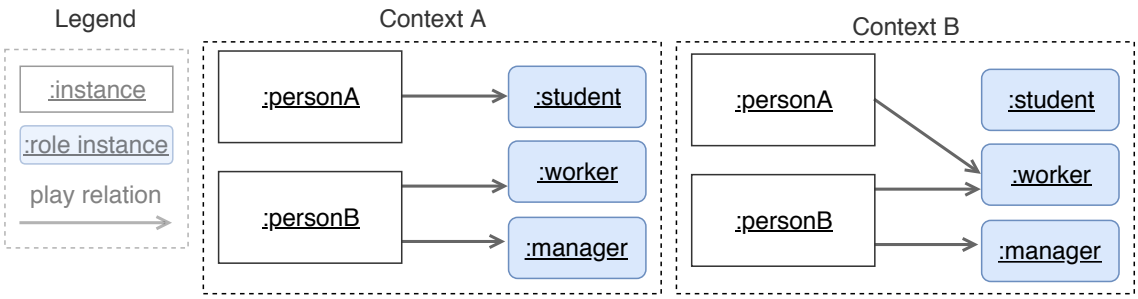


FIGURE 2.4 – Illustration de deux instances de la classe *Person*, *personA* et *personB*, qui jouent des rôles différents selon les contextes. Dans le contexte *A*, *personA* joue le rôle d'un étudiant tandis que *personB* joue les rôles de travailleur et de manager. Dans le contexte *B*, *personA* perd son rôle d'étudiant et gagne le rôle de travailleur. Les rôles sont ici des instances de rôles, qui sont dynamiquement associés aux objets. Lorsqu'une instance de *Person* acquiert ou perd un rôle, cette instance acquiert, altère ou perd du comportement et des états. Un objet peut jouer plusieurs rôles, et plusieurs objets peuvent partager le même rôle.

Un cas d'illustration classique est l'exemple d'une personne et des rôles qu'elle peut jouer durant sa vie. La Figure 2.4 montre deux objets *personne*, instances d'une classe *Personne*. Ces deux objets possèdent une structure ainsi qu'un comportement identiques. Telle quelle, une instance de *Personne* est dénommée *core-object*, ou *objet principal* [Fowler, 1997, Bäumer et al., 1998]. Il s'agit d'une instance *nue*, avec une identité propre et dont l'état et le comportement sont entièrements décrits dans la classe *Personne*. Steimann décrit ce *core-object* comme étant issu d'un type *naturel*, *"indépendant de l'existence de relations avec d'autres entités"* [Steimann, 2000]. Trois rôles sont représentés, par des instances de *Student*, *Worker* et *Manager*. Ces derniers représentent *"une perspective sur un objet existant"*, sous la forme *"d'extensions dynamiques de classes"*, et apportent une altération de l'état et du comportement de l'objet auquel ils sont associés [Graversen and Osterbye, 2003]. Les deux instances de personnes jouent alors des rôles différents, qui varient selon les contextes. Dans le contexte *A*, *personA* joue le rôle d'un étudiant, tandis que *personB* joue le rôle de travailleur et de manager. Dans le contexte *B*, *personA* change dynamiquement de rôle et n'est plus un étudiant mais un travailleur. L'acquisition ou la perte de rôles est dynamique, totalement dépendante du contexte, et préserve l'identité de l'objet affecté. Par exemple, la terminaison du rôle *student* pour l'objet *personA* n'a pas d'implication sur son identité en tant qu'instance de personne. En revanche, la perspective ou la vue sur l'objet est modifiée, car *personA* joue le rôle de *worker* dans le contexte *B*. Pour Steimann, jouer un rôle revient, pour un objet, à assumer différentes formes au cours de son existence [Steimann, 2000].

Les rôles pour l'adaptation dynamique. Les rôles dynamiques permettent d'aborder l'adaptation dynamique de comportement dans un système en cours d'exécution. Typiquement, l'acquisition d'un rôle permet d'obtenir ou d'altérer le comportement d'un objet. L'utilisation des rôles pour l'adaptation dynamique a donné lieu à de nombreux travaux, visant à supporter la variabilité de comportement dans un logiciel ou ciblant des applications particulières.

ROPE est un environnement pour les systèmes multi-agents [Becht et al., 1999], dans lequel les comportements de coopérations entre agents sont spécifiés dans des rôles, et peuvent varier dynamiquement.

Chameleon [Graversen and Osterbye, 2003], *ObjectTeams* [Herrmann, 2005] et *Rava* [He et al., 2006] sont des extensions de Java qui supportent les rôles dynamiques au niveau langage. *ObjectTeams* permet de regrouper des rôles dans des entités de premier niveau nommées *Teams*, qui représentent des contextes d'adaptation au sein desquels les rôles sont encapsulés. Lorsqu'une *Team* a été définie, instanciée et activée, tout appel à une méthode redéfinie par un des rôles de cette instance sera intercepté et sa version adaptée sera exécutée.

Piechnick et al. utilisent des rôles pour adapter des ports dans des systèmes à composants (voir section 2.1.3) [Piechnick et al., 2012]. Les ports des composants

jouent des rôles, et la variation des rôles permet de modifier les interfaces des composants et leur inter-connexions.

Schneider propose *Role4All* [Schneider et al., 2015a, Schneider et al., 2015b], qui met en œuvre des rôles pour adapter dynamiquement des modèles d'entrée de différents formalismes dans les systèmes de systèmes [Maier, 1998]. Les rôles permettent d'appliquer différentes vues sur les modèles, afin de pouvoir les interpréter et les simuler au sein du même environnement.

Le modèle *Epsilon* [Tamai, 1999, Kamina and Tamai, 2008] confine les rôles à un périmètre nommé *contexte*, au sein duquel les rôles collaborent. Les objets peuvent dynamiquement entrer ou sortir d'un contexte en assumant des rôles. *EpsilonJ* [Tamai et al., 2005, Monpratarnchai and Tetsuo, 2008] est un langage basé sur Java mettant en œuvre le modèle Epsilon. Ce dernier permet la définition de rôles qui implémentent et composent du comportement spécifique à des contextes. Le contexte est exprimé à haut niveau en tant que construction de langage, qui spécifie les rôles qui lui sont associés. Un opérateur dédié *bind* permet de lier un rôle à un objet. *NextEJ* [Kamina and Tamai, 2009] est une autre mise en œuvre du modèle *Epsilon*, variante d'*EpsilonJ*, comprenant la sûreté des types exprimée par le modèle et qui permet d'expliciter le périmètre d'activation des rôles (c'est-à-dire de leur contexte). Dans *NextEJ*, *bind* définit l'entrée dans le périmètre d'un contexte, et donc de son activation. Les rôles ne sont actifs que dans le périmètre explicite de l'activation de leur contexte. *EpsilonJ* et *NextEJ* supportent explicitement l'association de rôles à une granularité objet.

LyRT [Taing et al., 2016a, Taing et al., 2016b, Taing et al., 2017, Taing, 2017] est un environnement avec un modèle d'exécution dédié, qui introduit la notion de *liaison d'instance dynamique*. Les rôles et les objets sont dynamiquement associés dans une table de *lookup* pendant l'exécution du programme, et ces liens sont utilisés pour retrouver les méthodes des rôles que joue un objet. L'environnement d'exécution permet d'introduire de nouveaux rôles après le démarrage du système, permettant ainsi l'adaptation non-anticipée de comportement. L'exécution de comportement peut être encapsulée dans une transaction pour permettre au système d'effectuer une transition sécurisée entre deux versions du comportement. Un objet ne change alors de rôle qu'une fois qu'il a cessé toute utilisation du comportement engagé dans une transaction. *LyRT* fournit également un support explicite pour la réversibilité d'une adaptation, et peut revenir à une version antérieure de l'application en cas d'erreur survenant après une adaptation.

L'adaptation non-anticipée avec les rôles. Peu de solutions basées sur les rôles supportent l'adaptation non-anticipée de comportement. Certaines solutions reposent sur un autre paradigme pour pouvoir introduire de nouveaux rôles après le démarrage du programme, comme par exemple les composants [Piechnick et al., 2012] ou les aspects [Sanen et al., 2005].

Role4All [Schneider et al., 2015a, Schneider et al., 2015b] est basé sur Pharo Smalltalk [Black et al., 2010], qui est un environnement de *Live Programming*(voir section 2.1.2). Ce dernier permet alors des modifications non-anticipées du programme par définition, mais il ne s'agit pas d'une fonctionnalité particulière fournie par *Role4All*. En particulier, toutes ces solutions héritent des limitations de leur support au regard des propriétés recherchées pour l'adaptation non-anticipée (section 1.3.3). *LyRT* [Taing et al., 2016a, Taing et al., 2016b, Taing et al., 2017] est, à notre connaissance, la seule solution basée sur les rôles qui fournit un support dédié pour adapter un programme de manière totalement imprévisible.

Limites pour l'adaptation non-anticipée d'objets. Nous ne considérons ici que les solutions, très récentes au regard du domaine, capables d'adaptation non-anticipée. Nous considérons en particulier que *LyRT* représente l'état de l'art des possibilités pour l'adaptation non-anticipée avec les rôles. *LyRT* est dit *généraliste*, pouvant être mis en œuvre dans d'autres langages objets, même s'il n'en existe aujourd'hui qu'une implémentation Java.

Le premier aspect pouvant être discuté est la *sortie* du paradigme objet. L'utilisation et la programmation de rôles nécessite la manipulation de concepts de haut niveau pour définir du comportement parallèlement aux classes, comme les *contextes*, les *rôles*, etc. Au premier abord il s'agit d'un autre paradigme, notamment en terme de modèle mental pour un-e développeur-euse. Il y a cependant des arguments qui tempèrent cette affirmation. Tout d'abord, l'application dynamique des rôles en tant que perspectives (ou vues) sur des objets ne fait qu'achever une sorte de polymorphisme avancé, propriété fondamentale du paradigme objet. Il est toujours possible de considérer que la programmation par les rôles constitue surtout une extension de la programmation par objets, plutôt qu'un changement radical de paradigme. Ensuite, à l'instar des possibilités offertes par *LyRT*, il est possible d'exprimer les rôles et leurs contextes tout en restant strictement dans le paradigme objet. Les rôles sont alors définis en héritant d'une classe *Role* ou en implémentant une interface spécifique. Du point de vue utilisateur, si l'objectif est d'appliquer des rôles à des instances, leur définition correspond toujours à des classes qui sont instanciées pour donner des instances de rôles.

Le second aspect, cette fois limitant, est le manque de support pour le minage d'objet. Il n'existe pas de moyen trivial et immédiat de le mettre en œuvre avec les rôles. Concrètement, il n'est donc pas possible de retrouver des objets spécifiques pour appliquer une adaptation. *LyRT* stocke les objets de l'environnement d'exécution dans une table de *lookup*, et permet d'y référer pour l'association non-anticipée de rôles, mais cela ne concerne que les objets *stables* dans le système. En effet, les objets temporaires ou avec une durée de vie extrêmement courte sont collectés et éliminés par le gestionnaire de mémoire dès qu'ils ne sont plus utilisés. S'ils ne sont

pas associés à des rôles au moment précis de leur utilisation, l'adaptation ne peut pas se faire et par la suite ils n'existent plus dans la table de *lookup* de *LyRT*.

2.2.3 Programmation Orientée Contexte (COP)

Un contexte peut être défini comme suit [Hirschfeld et al., 2008] : *"Any information which is computationally accessible may form part of the context upon which behavioral variations depend."*. Un contexte peut donc dépendre de l'état interne du programme ou de ses objets, d'évènements ou de données externes du programme, ou encore de calculs et de transformations d'informations issues de différentes sources. La modélisation de comportements différents suivant les contextes est alors potentiellement complexe et verbeuse, et nécessite l'utilisation intensive d'instructions conditionnelles pour prendre en compte toutes les possibilités. Dans cette section, nous regroupons les approches qui considèrent et définissent une notion de *contexte* en tant que concept exprimant une variation comportementale.

2.2.3.1 *COP*

La *Programmation Orientée Contexte* (*COP*) permet la définition explicite de *contextes* pour exprimer et mettre en œuvre la variation comportementale d'un système [Hirschfeld et al., 2008, Salvaneschi et al., 2012a]. Lorsqu'un objet entre dans un contexte, une altération comportementale spécifique à ce contexte lui est appliquée, puis retirée lorsqu'il sort de ce contexte. Il est sémantiquement possible de définir du comportement relatif à un contexte donné, et les objets du programme sont alors capables de s'adapter aux différents contextes qu'ils rencontrent. Plusieurs langages COP ont été développés depuis le début de l'étude du domaine [Appeltauer et al., 2009, Salvaneschi et al., 2012a], comme par exemple *ContextL* pour CLOS [Costanza and Hirschfeld, 2005], *ContextS* pour Smalltalk [Hirschfeld et al., 2007] ou encore *ContextJ* pour Java [Appeltauer et al., 2011] et *ContextJS* pour Javascript [Lincke et al., 2011, Krahn et al., 2012]. Les contextes y sont représentés par une construction de langage de premier ordre nommée *Layer*, au sein de laquelle sont définies des méthodes représentant une variation comportementale spécifique à un contexte. Lorsqu'un objet entre dans un contexte, ou *layer*, et que ce dernier est *actif*, le comportement de l'objet est modifié par celui défini dans la *layer*. Dans les paragraphes suivants, nous discutons rapidement les notions de *layer* et de *Scope* (périmètre d'activation d'une *layer*).

Layer. Les *layers* sont des concepts de haut niveau et de premier ordre, accessibles sous forme de construction de langage. Elles peuvent être instanciées et stockées dans des variables. Comme une classe, une *layer* est déclarée avec un nom et un corps, qui contient des méthodes. Elles sont dites *modularisées*, c'est-à-dire

que le code mettant en œuvre du comportement contextuel est strictement séparé du code du programme de base. Il y a typiquement deux cas de figure : soit les *layers* sont définies au sein des classes, il s'agit alors d'une partie indépendante du comportement de la classe, soit les *layers* sont définies de manière externe aux classes [Appeltauer et al., 2009, Salvaneschi et al., 2012a]. L'activation d'une *layer* permet de rendre active la variation comportementale qu'elle met en œuvre, et qui s'applique sur des objets particuliers. Les conditions d'activation d'une *layer* ainsi que les objets qui sont concernés par l'adaptation forment un périmètre d'activation, nommé *scope*.

Scope. Le périmètre d'activation, ou *scope*, détermine quand une *layer* est active et pour quels objets. Il existe différentes stratégies de *scoping* [Salvaneschi et al., 2012a]. L'activation basée sur le flot de contrôle, ou *Dynamically Scoped Activation*, restreint l'activation d'une *layer* à un bloc de code. Tous les objets exécutant du comportement au sein de ce flot de contrôle, y compris par appels de méthode imbriqués, sont soumis à la variation de comportement mise en œuvre par la *layer*. Cette dernière est désactivée lorsque l'exécution sort du flot de contrôle défini par le bloc de code. La composition et l'activation de *layers* peuvent être conditionelles [Appeltauer et al., 2010, Ramson et al., 2017], ou encore restreintes à des objets particuliers [Salvaneschi et al., 2012a]. L'activation et la transition entre deux *layers* peuvent être basées sur des évènements [Kamina et al., 2011]. La stratégie d'activation peut également être *implicite* [Von Löwis et al., 2007], et une *layer* peut alors être déclarée *active* ou *inactive*. L'adaptation d'une méthode est rendue effective par la vérification dynamique des *layers* actives lorsque cette méthode est invoquée. D'une autre manière, *ContextJS* [Lincke et al., 2011] ouvre la stratégie de *scoping* et son implémentation, et il est possible d'étendre le *scoping* par de nouvelles stratégies définies par l'utilisateur-rice.

2.2.3.2 Mécanismes en marge des technologies *COP*

Les techniques et mécanismes qui suivent ne font pas à proprement parler partie de la *Programmation Orientée Contexte*, mais se situent en marge de cette dernière. Il s'agit de solutions s'inspirant des concepts *COP* ou les mettant en œuvre dans des cas d'utilisation particuliers.

Adaptation basée sur les groupes. L'adaptation basée sur les groupes est un ensemble de techniques émergentes qui visent à sélectionner des objets pour leur appliquer une adaptation de comportement [Rein et al., 2017]. Rein et al. décrivent et analysent l'adaptation basée sur les groupes comme étant la combinaison d'un mé-

canisme de sélection d'objets, qui peut être explicite ou implicite, et d'un mécanisme d'adaptation pour adapter ces objets [Rein et al., 2017].

Dans *ContextErlang* [Salvaneschi et al., 2012b] (spécification explicite), l'adaptation est activée et composée au travers de messages que se transmettent les agents du système. L'envoi des messages d'activation est directement mis en place dans le comportement des agents, et la granularité de l'adaptation se situe au niveau de l'agent lui-même. La sélection d'agents se fait donc par envoi de messages entre et aux différents agents. *Reactive Object Queries* [Lehmann et al., 2016] (spécification implicite) sont des requêtes qui permettent de créer et de maintenir automatiquement des collections d'objets satisfaisant des propriétés particulières. Les objets peuvent entrer ou sortir dynamiquement des groupes en fonction du critère de sélection exprimée dans une requête et de l'évolution de leur état. Les *Context Groups* du langage *ServalCJ* [Kamina et al., 2016] regroupent des instances qui seront adaptées par la même modification comportementale. Les objets peuvent dynamiquement souscrire aux groupes, en appelant directement une méthode de l'objet représentant le groupe. Ces souscriptions aux groupes doivent cependant être anticipées avant le démarrage du programme.

L'adaptation peut se faire de différentes manières, et notamment par l'utilisation de mécanismes similaires à (ou issus de) *COP*. Rein et al. relèvent cependant que ces techniques sont plus adaptées aux langages à objets sans classes [Rein et al., 2017], c'est-à-dire typiquement aux langages à prototypes [Dony et al., 1998]. Dans les langages objets, cela permet cependant de regrouper les objets de manière hétérogène, et de les adapter indépendamment de leur classe.

À l'instar des mécanismes *COP*, l'adaptation basée sur les groupes est globalement soit anticipée soit invasive. La spécification des groupes d'objets, explicite ou implicite, est pré-déterminée avant l'exécution du programme. Les solutions étudiées ne détaillent pas de support particulier pour un quelconque aspect non-anticipé, et se concentrent sur l'aspect dynamique de la constitution des groupes d'objets. Certaines solutions permettent la spécification ou la modification non-anticipée de groupes, par exemple via le *Live Programming*, mais nécessitent d'insérer du code et sont donc invasives.

Context-Traits. Un *trait* [Ducasse et al., 2006] est originellement constitué de méthodes formant un ensemble cohérent entre elles, et pouvant être réutilisé par plusieurs classes différentes. *Context-Traits* [González et al., 2013] met en avant l'utilisation et la composition dynamique de *traits* en tant qu'unités d'adaptation comportementale. Les *traits* affectent ici le comportement du programme à une granularité instance, mais leur composition amène des conflits qui doivent être résolus avant leur application. Des politiques de composition permettent de résoudre ces conflits automatiquement pendant l'exécution du programme. Un autre inconvénient lors

de l'utilisation des *traits* est la nécessité de mettre en œuvre du code dédié supplémentaire, car un *trait* peut requérir des interfaces qui doivent être implémentées quelque part. Cela pose une contrainte additionnelle et des dépendances de code sur la conception du logiciel.

2.2.3.3 Limites pour l'adaptation non-anticipée d'objets

La programmation orientée contexte fait figure d'exception dans l'ensemble des solutions étudiées par cet état de l'art, en ce fait qu'elle ne permet pas réellement l'adaptation non-anticipée de comportement. Les variations comportementales et leurs périmètres d'activation sont en général anticipés avant le démarrage du programme, à l'exception de solutions comme *Context-Traits* [González et al., 2013] ou encore de *ContextS* [Hirschfeld et al., 2007] − ce dernier pouvant reposer sur le support de *Live Programming* fourni par Smalltalk pour les interactions avec le programme en cours d'exécution. D'autre part, l'utilisation des *layers* comme construction de haut niveau en parallèle des classes impose des contraintes en terme de flexibilité, car il y a nécessité d'utiliser un langage dédié ou des constructions de langage particulières. Mais *COP* permet cependant de caractériser et de contextualiser fortement la variation comportementale, ce qui justifie l'évocation de la technique. Il y a séparation, par les *layers*, du code de base du programme et du code représentant les adaptations que doivent subir les objets, garantissant à l'instar de l'*AOP* une séparation nette des préoccupations. Le *scoping* permet de manière implicite de restaurer le comportement d'un objet, lorsqu'une *layer* est désactivée. Les stratégies d'activation basées sur le flot de contrôle, bien qu'intrusives, permettent de sélectionner implicitement − et de manière purement statique − les objets qui seront affectés par une adaptation. Les mécanismes d'adaptation basée sur les groupes permettent de définir plus finement des ensembles d'objets à adapter, au-delà de leur simple appartenance à une classe particulière [Rein et al., 2017], mais la définition de groupes doit être anticipée avant le démarrage du programme.

2.2.4 Discussion

Plusieurs propriétés ne sont pas satisfaites par les grands paradigmes de programmation étudiés pour l'adaptation non-anticipée de comportement. L'évaluation de ces paradigmes de programmation étudiés est résumée dans la table 2.2.

Le minage d'objet constitue un point faible de toutes les solutions étudiées. C'est un aspect en demi-teinte pour *COP*, qui ne l'achève pas totalement non plus mais qui fournit tout de même une contextualisation de l'adaptation pour les objets. Toutes les instances qui *entrent* dans le périmètre d'activation d'une *layer* sont en quelque sorte *sélectionnées* pour être adaptées. Les mécanismes d'adaptation basée

sur les groupes fournissent plus de finesse dans la sélection d'objets, mais il n'est cependant pas possible de les utiliser de manière non-anticipée et non-invasive.

Deux aspects non évoqués pour *COP* sont l'aspect intrusif des *scopes* basés sur le flot de contrôle, car ils nécessitent d'insérer des instructions d'activation des *layers* dans le code de base, et la réversibilité des adaptations. La réversibilité est implicite, car la désactivation d'une *layer* implique le retour de l'objet à son comportement non-adapté − ou au comportement correspondant aux *layers* actives pour cet objet.

La flexibilité, telle que nous l'entendons pour le déverminage, n'est que partiellement achevée pour les rôles et pour *COP*. En ce qui concerne les *rôles*, la solution la plus avancée (*LyRT*) impose l'utilisation de son environnement d'exécution. Par exemple, *LyRT* force à instancier les objets au travers de son registre d'objets, afin de pouvoir les référencer dans sa table de *lookup* et plus tard de les adapter. *COP* force l'utilisation d'un langage (ou d'une extension) et l'utilisation de plusieurs concepts particuliers lors du développement du programme (*layers*, *scopes*).

Globalement, l'*AOP*, *COP* et les rôles préservent l'identité des objets adaptés. Dans le cas des rôles il s'agit d'une propriété intrinsèque du paradigme, tandis que l'*AOP* et *COP* mettent en œuvre des techniques de méta-programmation (voir section 2.3) qui n'influent pas sur l'identité de l'objet adapté − ou rendent transparente toute transition de l'objet vers une autre structure.

● Totalement supporté ◑ Partiellement supporté ○ Non supporté

	Granularité objet	Mimage d'objets	Réversibilité	Identité préservée	Flexibilité	Non-intrusivité	Paradigme objet
Programmation Orientée Aspects	●	○	●	●	●	●	○
Programmation par les Rôles	●	◑	●	●	◑	●	◑
*Programmation Orientée Contexte	●	◑	●	●	◑	◑	○

* Avec les mécanismes d'adaptation basée sur les groupes.

TABLE 2.2 – Positionnement des approches de type paradigmes de programmation pour l'adaptation non-anticipée.

Note de mise en œuvre. Les aspects constituent une possibilité intéressante sur laquelle reposer pour implémenter des mécanismes d'adaptation ou de débogage.

Certains dévermineurs utilisent ainsi de manière transparente des techniques basées sur les aspects, c'est également le cas de certains langages orientés contexte.

2.3 Approches réflexives

La reflexivité est "la capacité d'un programme à manipuler comme des données des éléments représentant l'état du programme pendant sa propre exécution" [Bobrow et al., 1993]. Bobrow et al. définissent deux aspects de la réflexivité, qui sont l'introspection et l'intercession. L'introspection est la capacité du programme à s'observer lui-même, et l'intercession sa capacité à se modifier lui-même d'un point de vue sémantique (son comportement). Le mécanisme permettant de transformer des informations relatives à l'exécution en données manipulables par le programme est nommé *réification*.

La réflexivité peut être divisée en deux catégories : la réflexivité structurelle et la réflexivité comportementale [Malenfant et al., 1996]. La réflexivité structurelle concerne la réification des données structurelles du programme, tandis que la réflexivité comportementale concerne la réification de données sémantiques sur l'exécution du programme. La réification du nom de la classe d'un objet est un exemple de réflexivité structurelle. La modification du comportement de réception de message dans la classe d'un objet est un exemple de réflexivité comportementale.

Les *méta-objets* sont des objets représentant des entités du langage, et qui définissent son comportement [Kiczales et al., 1991]. Ce sont par exemple des *"classes, méthodes, fonctions, [...] qui représentent des fragments du programme"* [Kiczales et al., 1991]. Les objets communiquent naturellement avec leurs méta-objets par une interface, qui lorsqu'elle est exposée porte le nom de protocole à méta-objets, ou *MOP* [Tanter, 2004]. Les *MOPs* permettent le contrôle des méta-objets. Dialoguer avec les méta-objets au travers des *MOPs* rend possible la modification dynamique du comportement du langage et du programme. Une définition de Kiczales et al., couramment reprise, est la suivante :

> *"Metaobject protocols are interfaces to the language that give users the ability to incrementally modify the language's behavior and implementation, as well as the ability to write programs within the language."* [Kiczales et al., 1991]

La modification et l'instrumentation du comportement du langage par des mécanismes réflexifs est alors un moyen de choix pour influer sur le comportement du programme. Dans cette section, nous évaluons les techniques basées sur la réflexivité pour l'adaptation non-anticipée de comportement dans les programme à objets. Aucun de ces mécanismes ne supporte le minage d'objets, cette propriété n'est donc pas évaluée pour les solutions étudiées.

2.3.1 Classes anonymes, *wrappers*, *proxies*

Ces techniques et mécanismes consistent à modifier le comportement d'un programme en modifiant sa structure − en migrant les objets vers des classes mettant en œuvre une adaptation − ou en encapsulant le comportement ou les objets dans des *coquilles*. Ces *coquilles* interceptent et contrôlent les évènements venant de *l'extérieur* de la *coquille* − typiquement des messages − et décident ensuite comment traiter ces évènements pour produire une variation comportementale.

Sous-classes anonymes. Une sous-classe anonyme est une classe spécifique qui est dynamiquement insérée entre un objet et sa classe d'origine [Foote and Johnson, 1989, Hinkle et al., 1993, Ducasse, 1999] . L'objet est alors migré vers cette classe anonyme, et cette dernière prend comme super-classe la classe d'origine de l'objet. Les méthodes de la classe d'origine de l'objet peuvent alors être redéfinies et adaptées dans la classe anonyme. Lorsqu'un message est envoyé à l'objet, le *lookup* démarre d'abord dans la classe anonyme. Si la méthode correspondant au message a été redéfinie, alors elle est exécutée. Sinon le *lookup* échoue et continue dans sa super-classe, c'est-à-dire la classe d'origine de l'objet où le comportement d'origine sera retrouvé et exécuté. L'utilisation des classes anonymes permet alors d'adapter individuellement le comportement des objets du programme [Ducasse, 1999]. Hinkle et al. appellent ces classes des *Lightweight Classes* [Hinkle et al., 1993], et s'en servent pour déverminer des objets. Leur mise en oeuvre repose sur une spécialisation du compilateur Smalltalk [Goldberg and Robson, 1983]. La possibilité de générer une sous-classe anonyme à partir d'une classe du système est désormais intégrée dans les langages modernes issus de Smalltalk comme Pharo [Black et al., 2010].

Un objet adapté avec les sous-classes anonymes ne change pas d'identité au sens où sa structure est préservée (états, méthodes, références) et que l'opération de migration est transparente pour les autres objets qui référencent éventuellement ce dernier. En revanche, la flexibilité de la solution est limitée sur deux points. Premièrement, elle repose sur la copie brute des méthodes de la classe d'origine vers la nouvelle classe anonyme. Il n'y a pas d'automatisation pour faciliter l'instrumentation de code et il s'agit d'une fonctionnalité d'assez bas niveau. Il manque des abstractions pour manipuler la solution dans le cadre de l'adaptation non-anticipée de comportement. Les sous-classes anonymes pourraient cependant servir de base pour l'implémentation d'une solution de plus haut niveau. Secondement, la technique est mise en œuvre avec Smalltalk [Goldberg and Robson, 1983], et repose sur certaines facilités réflexives du langage, ce qui pose des questions sur sa portabilité. Il n'y a pas non plus de support explicite pour renverser ou modifier une adaptation de comportement. Il est toutefois possible de re-migrer l'objet vers sa classe d'origine pour annuler l'adaptation courante de l'objet (*réversibilité*). En revanche

pour modifier une adaptation, il faut créer une nouvelle sous-classe anonyme et migrer l'objet vers cette dernière ou supprimer manuellement les méthodes de la classe anonyme courante de l'objet. Nous considérons donc que la propriété de *flexibilité*, nest pas complètement atteinte.

Method Wrappers. Un *wrapper* permet d'insérer du comportement avant, après ou à la place du comportement d'origine d'une méthode [Brant et al., 1998, Ducasse, 1999]. Il s'agit d'une méthode spéciale qui vient remplacer la méthode d'origine par un *enrobage*, ou *wrapper*, qui est du code mettant en œuvre le comportement inséré. Lorsqu'un appel à la méthode *wrappée* est effectué, le *wrapper* est exécuté en lieu et place, et peut éventuellement invoquer la méthode originale. Cette dernière ainsi que son code source sont préservés, et le *wrapper* peut être désinstallé à tout moment. La technique permet également de conserver la visualisation du code d'origine des méthodes instrumentées dans les outils de l'environnement de développement. Nous la considérons donc comme étant non-intrusive. Cependant, les méthodes se situant dans des classes, *wrapper* une méthode pour adapter aura pour effet de modifier le comportement de toutes les instances de la même classe. Au sens strict, les *wrappers* ne permettent donc pas d'adapter avec une granularité objet. La réversibilité n'est que partiellement satisfaite, car s'il est possible de désinstaller un *wrapper*, cela se fait avec une granularité classe et donc il n'y a pas de support explicite pour désinstaller un *wrapper* d'un objet spécifique. La préservation de l'identité des objets est implicite, car ce sont les classes qui sont modifiées et cela impacte toutes leurs instances. La mise en œuvre des *wrappers* [Brant et al., 1998] permet la migration de toutes les instances vers la nouvelle version de leur classe de manière transparente, car elle repose sur Smalltalk [Goldberg and Robson, 1983] qui le permet nativement. Les *wrappers* peuvent être combinés avec la technique des sous-classes anonymes, afin d'obtenir une granularité objet [Brant et al., 1998].

Techniques à base de *proxy*. Un *proxy* est un objet qui s'insère de manière transparente entre un objet et les autres objets utilisateurs de ce dernier [Gamma et al., 1993, Ducasse, 1999, Fowler, 2004]. Le *proxy* remplace donc l'objet, présente la même interface, et ses utilisateurs *pensent* toujours converser avec cet objet d'origine. Tous les messages envoyés à l'objet d'origine sont alors interceptés par le *proxy* qui décide de la manière de traiter ces messages. Le comportement de l'objet est contrôlé par le *proxy*, qui peut alors mettre en œuvre des variations comportementales. Les mécanismes à base de *proxy* permettent d'aller plus loin que simplement se substituer à un objet. Par exemple en Java [Eugster, 2006], un *proxy* peut remplacer une classe. Les interfaces pouvant être interceptées par le *proxy* doivent cependant être déclarées avant le début de l'exécution du programme. La contrainte de devoir prévoir de manière statique les interfaces à adapter ne permet

pas de considérer cette solution pour l'adaptation non-anticipée. Les langages dynamiques permettent de lever cette limitation. Le *proxy* est utilisé en Javascript pour mettre en œuvre des mécanismes d'intercession [Van Cutsem and Miller, 2010]. Il est par exemple possible de remplacer une fonction par un *proxy* et d'en contrôler l'accès. *Ghost* [Peck et al., 2015] pour Pharo [Black et al., 2010] permet de remplacer un objet, une classe ou une méthode par un *proxy*. Dans *Ghost*, l'utilisateur-rice peut sélectionner de manière très fine les messages à intercepter.

Deux problèmes majeurs se posent avec les mécanismes de *proxy* pour l'adaptation non-anticipée. Le premier est que l'identité de l'objet est perdue, car cet objet est remplacé par un *proxy*. Tous les autres objets du système dialoguent alors avec le *proxy* et non plus avec l'objet. Le second est le problème du *self* [Lieberman, 1986]. Concrètement lorsqu'un objet est remplacé par un *proxy*, ce dernier intercepte tous les messages envoyés à l'objet. Mais les messages envoyés à la variable spéciale *self* ne sont pas interceptés par le *proxy*. Tous les messages envoyés à *self* lorsqu'un objet exécute une méthode − suite à la transmission d'un message intercepté ou d'un appel à partir du *proxy* − passeront au travers du mécanisme de *proxy*. Il s'agit d'une limitation sévère pour l'adaptation non-anticipée de comportement, et qui ne satisfait pas notre propriété de flexibilité. Il est possible de régler ce problème avec *Ghost*, en utilisant un *proxy* de classe et en instrumentant le *lookup* [Peck et al., 2015]. La responsabilité de la mise en œuvre d'une solution repose cependant sur les développeur-euse-s. Une extension pour *Ghost* prend en compte ce problème et fournit une interface pour muter un objet avec un *proxy* [Kudriashov, 2016]. Mais cela nécessite de définir, pour chaque type de *mutation*, la liste des messages qui ne doivent pas être interceptés comme par exemple certains méta-messages ou des interfaces spécifiques de l'objet qui ne doivent pas être altérées. Ce sont à nouveau des contraintes qui rendent la solution peu compatible avec notre définition de la flexibilité pour l'adaptation non-anticipée.

Les propriétés de réversibilité et de non-intrusivité sont atteintes, respectivement parce qu'il existe des interfaces pour *dé-proxy-fier* un objet et parce que l'utilisation d'un *proxy* ne nécessite pas d'injection de code dans la classe d'un objet.

Discussion. La table 2.3 résume le positionnement des approches réflexives étudiées dans les paragraphes précédents. La combinaison des *wrappers* et des sous-classes anonymes permet d'obtenir des propriétés intéressantes. Le problème de la réversibilité de l'adaptation pourrait être résolu par des moyens automatiques pour *dé-wrapper* toutes les méthodes d'un objet en une seule fois et le re-migrer vers sa classe d'origine. Il manque pour cela des abstractions de plus haut niveau pour définir une adaptation au niveau d'un objet, plutôt que de *wrapper* des méthodes une par une.

● Totalement supporté ◑ Partiellement supporté ○ Non supporté

	Granularité objet	Minage d'objets	Réversibilité	Identité préservée	Flexibilité	Non-intrusivité	Paradigme objet
Sous-classes anonymes	●	○	●	●	◑	●	●
Method Wrappers	○	○	◑	●	●	●	●
Wrappers + Sous-classes anonymes	●	○	◑	●	◑	●	●
Techniques à base de proxy	●	○	●	○	○	●	●

TABLE 2.3 – Positionnement des approches réflexives à base de restructuration et d'encapsulation pour l'adaptation non-anticipée.

2.3.2 Protocoles à méta-objets et méta-objets

Les solutions étudiées dans cette section utilisent la réflexion comportementale pour modifier le fonctionnement du programme. Ces techniques utilisent les méta-objets et leurs protocoles (les *MOPs* [Kiczales et al., 1991]) pour intercepter, rediriger et instrumenter les messages envoyés à des objets.

MetaclassTalk. *MetaclassTalk* est une extension de Smalltalk [1] dans laquelle les classes jouent le rôle de méta-objets qui contrôlent directement la manière dont leurs instances accèdent à leur état et exécutent leur comportement [Bouraqadi-Saâdani et al., 1998, Bouraqadi Saadani, 1999, Bouraqadi, 2000]. Les classes de *MetaclassTalk* contrôlent donc l'exécution des objets, comme l'envoi et la réception de messages ainsi que la sémantique du *lookup*. Ces opérations sont mises en œuvre dans les méta-classes, dont les classes sont instances. Par exemple lorsqu'un objet reçoit un message, une méthode dédiée à la réception de message est automatiquement appelée dans la classe de l'objet − qui est le seul et unique méta-objet de l'objet. Cette méthode réifie, entre autres, l'objet receveur du message, l'objet à l'origine du message, les arguments du message et la classe de démarrage du lookup. Le contrôle appliqué par cette méthode à la réception de message est défini dans la méta-classe dont la classe est instance. Il est alors possible de programmer des méta-classes qui spécialisent ou redéfinissent complètement le comportement des mécanismes du langage (par exemple le *lookup*) pour mettre en œuvre des variations comportementales non-anticipées.

1. La dernière mise en œuvre a été effectuée avec Squeak [Nierstrasz et al., 2009b]

Il est possible de changer le comportement d'un objet spécifique en remplaçant son méta-objet (c'est-à-dire sa classe) par un autre méta-objet (c'est-à-dire une autre classe) qui contrôle différemment son exécution. La nouvelle classe doit alors être instance d'une nouvelle méta-classe qui définit le contrôle d'exécution (par exemple une instrumentation du *lookup*). Mais la définition du contrôle dans les méta-classes s'applique avec une granularité au niveau classe. Toutes les méthodes de la classe seront donc soumises au même contrôle d'exécution, et il n'est pas possible d'adapter une partie restreinte du comportement d'un objet. Pour cela, l'instrumentation dans les méta-classes doit prendre en compte les méthodes qui doivent être adaptées et appliquer un filtrage. Cela signifie que c'est à l'utilisateur-rice de fournir une classe spécifique pour chaque objet qui doit changer de comportement, puis de développer une méta-classe dédiée avec une instrumentation manuelle du comportement d'exécution pour filtrer les méthodes à adapter. Nous considérons donc que la propriété de flexibilité n'est pas satisfaite car *MetaclassTalk* ne fournit pas d'abstractions permettant de faciliter l'unicité de ces couples classes/méta-classes pour adapter le comportement d'objets uniques dans le système. *MetaclassTalk* utilise les *Method Wrappers* pour mettre en œuvre le contrôle des méta-classes sur les classes. Pour cela les classes de *MetaclassTalk* doivent être compilées avec un compilateur dédié, ce qui pose à nouveau une contrainte de flexibilité. La propriété de granularité objet n'est que partiellement satisfaite selon nos critères d'évaluation, car elle se fait sous les contraintes énumérées ci-dessus.

Les propriétés de réversibilité et de non-intrusivité sont en revanche atteintes. Pour annuler l'adaptation il suffit de ré-associer la classe d'origine à l'objet modifié, tandis que l'adaptation se fait par instrumentation au niveau des méta-classes et non pas au niveau des méthodes du programme. Bien que les objets changent de classe pendant l'adaptation, il s'agit d'un mécanisme géré par *MetaclassTalk* et faisant partie intégrante des possibilités offertes par le système. Nous considérons alors que les objets conservent leur identité malgré leur possible changement de structure.

Iguana/J. *Iguana* [Gowing and Cahill, 1996] est une architecture réflexive générique pour la réflexivité comportementale. *Iguana/J* [Redmond and Cahill, 2000, Redmond and Cahill, 2002] est une mise en œuvre Java du modèle *Iguana*. *Iguana/J* permet de définir des groupes de méta-objets contrôlés par des protocoles à méta-objets (*MOPS*). Les *MOPs* forment des ensembles cohérents au regard des méta-objets qu'ils permettent de contrôler. Les classes et les objets du programme peuvent être individuellement associés à des *MOPs*, qui peuvent être dynamiquement définis, chargés dans le programme et connectés à des objets. Le *MOP* d'une classe ou d'un objet représente son comportement. L'adaptation non-anticipé du comportement du programme se fait alors par l'association dynamique de *MOPs* à des classes et à des objets du programme. Par exemple, il est possible de définir un *MOP* qui adapte le comportement de base d'un objet. Cela nécessite d'écrire

du code mettant en œuvre du comportement non-fonctionnel, pour spécifier l'interception et la redirection d'une méthode. Ces opérations sont implémentées dans le *MOP* en charge de fournir la variation comportementale pour l'objet ciblé. Dans ce cas, l'interception n'a lieu que pour les objets adaptés − c'est-à-dire les objets partageant le même *MOP* − et pas pour les autres entités du programme.

Iguana/J permet donc d'adapter des objets uniques, par l'association dynamique de protocoles à méta-objets. Ces (dés)associations dynamiques de *MOPs* n'influent pas sur l'identité de l'objet et ne nécessitent pas l'introduction d'instructions particulières dans le code de base du programme. Enfin, la dés-association d'un *MOP* permet d'annuler l'adaptation de comportement ou de modifier cette adaptation.

La propriété de flexibilité n'est cependant pas satisfaite. Premièrement, la mise en œuvre d'adaptation comportementale nécessite de penser et de développer du comportement non-fonctionnel, par exemple le comportement de redirection d'une méthode dans le *MOP*. Secondement, *Iguana/J* repose sur une extension de la machine virtuelle Java. Cela pose une contrainte forte sur l'environnement d'exécution du programme, qui doit donc s'exécuter sur cette machine virtuelle.

Talents. Les *Talents* [Ressia et al., 2014] sont des unités de comportement réutilisables et composables pouvant être dynamiquement associées à des objets. Ce sont concrètement des objets qui définissent du comportement, c'est-à-dire des méthodes. Un objet spécifique peut acquérir ou perdre dynamiquement un *talent* pendant l'exécution du programme, et de ce fait acquérir ou perdre du comportement. Un *talent* peut également exprimer des restrictions de comportement, par exemple l'interdiction d'accès à une méthode. Associé à un objet, ce dernier ne pourra plus exécuter la méthode spécifiée comme *interdite* par le *talent*. Les *Talents* vont plus loin que de la simple modification comportementale, et permettent aussi aux objets d'acquérir ou de perdre des états. Pour adapter un objet, un-e développeur-euse doit d'abord instancier un *talent* et modéliser l'adaptation de comportement. Pour adapter une méthode particulière, cela consiste concrètement à définir une nouvelle méthode et à la compiler dans l'instance du *talent*. Il faut ensuite associer ce *talent* à l'objet à adapter. Lorsque l'objet adapté recevra le message correspondant à la méthode adaptée, le *lookup* sera effectué dans le *talent* plutôt que dans la classe de l'objet. La méthode adaptée présente dans le *talent* sera retrouvée et exécutée à la place du comportement d'origine. Il n'y a pas d'abstraction permettant de manipuler ou de contrôler des adaptations de comportement, ce qui fait des *Talents* une solution de bas niveau pour l'adaptation non-anticipée.

Les *Talents* sont réversibles car il suffit de dés-associer un *talent* pour annuler une adaptation. Ils sont non-intrusifs, car ce sont dans les faits des méta-objets. Le code adapté est défini dans ces méta-objets et le comportement du programme est modifié par association des objets qui le composent à des *talents* − le code de base

n'est donc jamais impacté. L'association d'un *talent* à un objet ne perturbe pas non plus l'identité de l'objet.

La propriété de flexibilité n'est pas totalement satisfaite pour deux raisons. Tout d'abord, les *Talents* sont mis en œuvre sur la base de l'architecture réflexive *Bifröst* [Ressia et al., 2010, Ressia et al., 2012]. *Bifröst* fournit des capacités réflexives structurelles et comportementales qui permettent l'adaptation non-anticipée à granularité très fine (ici l'objet). *Bifröst* est mis en œuvre dans Pharo [Black et al., 2010], et est complètement intégré à l'environnement réflexif du langage. Pour pouvoir mettre en œuvre la totalité de la solution dans un autre langage, il faut que ce dernier possède des capacités réflexives au moins équivalentes à celles de *Bifröst*. Cette affirmation se tempère si seulement certains types de modifications comportementales sont recherchés, par exemple l'ajout et la modification non-anticipée de méthodes. Il est alors possible d'implémenter un modèle incomplet des *talents* dans des environnements réflexifs moins complets que *Bifröst*. Le second problème de flexibilité concerne la composition des *talents*. Les *talents* peuvent être composés mais les conflits doivent être résolus manuellement par l'utilisateur-rice. L'utilisation de la composition n'est cependant pas obligatoire dans la conception d'adaptations, mais cela peut devenir un facteur limitant.

Reflectivity. La réflexion comportementale partielle, ou *Partial Behavioral Reflection* [Tanter et al., 2003], est un protocole à méta-objets avec une granularité très fine permettant de sélectionner de manière précise quelles entités réifier et à quel moment de l'exécution du programme. Par exemple, il est possible de réifier l'accès en écriture à une variable d'instance pour une classe donnée et d'effectuer une opération avant ou après l'écriture dans la variable. La réflexion comportementale partielle non-anticipée, ou *Unanticipated Partial Behavioral Reflection* [Röthlisberger et al., 2008], permet d'effectuer ces opérations réflexives de manière complètement non-anticipée.

La réflexivité au niveau *sous-méthode*, ou *Sub-method Reflection* [Denker et al., 2007], apporte des capacités réfléxives au niveau des sous-éléments d'une méthode, par exemple l'accès à une variable locale en lecture ou en écriture, ou encore un envoi de message. Cette dernière est basée sur des annotations de l'arbre de syntaxe abstraite du programme, ou AST. Un AST est une représentation minimale du code source sous forme d'arbre, éludant toutes les spécificités syntaxique du langage [Jones, 2003]. Dans les langages objets comme Smalltalk [Goldberg and Robson, 1983] ou Python [Python, 2017], les nœuds de l'AST sont des objets manipulables pendant l'exécution du programme. La réflexivité au niveau *sous-méthode* permet d'annoter l'AST d'une méthode avec des instrumentations sans modifier le code source de la méthode, comme par exemple compter tous les accès à une variable dans une méthode

spécifique [Denker et al., 2007].

Reflectivity [Denker, 2008, Nierstrasz et al., 2009a] est une mise en œuvre de la réflexivité comportementale partielle non-anticipée et de la réflexivité au niveau *sous-méthode* au sein d'un même modèle. *Reflectivity* fournit une abstraction nommée *MetaLink* − ou *méta-lien* − qui annote l'AST d'une méthode pour définir quelles entités doivent être réifiées, par exemple l'objet qui reçoit un message, et quand elles doivent être réifiées, par exemple avant la réception du message. Un *méta-lien* spécifie, pour un nœud annoté dans une méthode, une action à effectuer sur un méta-objet, une sélection d'éléments du contexte d'exécution de nœud à réifier et une condition d'exécution du *méta-lien*. Lors de l'exécution du code représenté par les nœuds annotés, la condition du *méta-lien* est évaluée et ce dernier est exécuté si la condition est satisfaite. L'exécution du *méta-lien* provoque la réification des entités demandées qui sont passées au méta-objet, qui exécute le comportement spécifié par le *méta-lien*. Il est alors possible de modifier le comportement d'une méthode avec une granularité très fine. La méthode peut être adaptée avec du comportement placé avant, après ou en remplacement complet de la méthode ou d'une de ses sous-parties.

Reflectivity annote des nœuds de l'AST, par définition liés à des méthodes et donc à des classes. Lorsqu'un *méta-lien* est installé sur une méthode, toutes les instances de la classe qui définit la méthode sont affectées par l'instrumentation. Il est possible d'atteindre une granularité objet en posant une condition dans le *méta-lien*, afin de ne l'exécuter que pour un objet ciblé. Toutes les instances de la classe concernée évalueront la condition pour savoir si le *méta-lien* les concerne. C'est un mécanisme qui devient difficile à utiliser dès qu'il faut affecter plusieurs objets avec le même *méta-lien*, et si cet ensemble d'objets peut varier dynamiquement. Nous considérons donc que la granularité objet n'est pas atteinte par *Reflectivity*. Pour les mêmes raisons, la réversibilité est limitée à la désinstallation de *méta-liens* avec une granularité classe, et non pas objet. Nous considérons cette propriété comme partiellement satisfaite. En ce qui concerne la flexibilité de la solution, nous ne la considérons que partiellement satisfaite. La définition et l'installation de *méta-liens* sont des opérations de plutôt bas niveau par rapport à l'adaptation comportementale d'un objet, et requièrent une connaissance avancée du système pour manipuler la syntaxe abstraite du programme.

Reflectogram. Le *Reflectogram* [Tanter et al., 2003] est une illustration visuelle des sauts entre le niveau de base, où le code du programme est exécuté, et le méta-niveau où des opérations réflexives sont exécutées. Papoulias *et al.* réifient le *Reflectogram* sous la forme d'un méta-objet [Papoulias et al., 2017] qui expose une API pour contrôler les opérations réflexives du programme. Cette API permet d'intercepter des évènements de l'exécution du programme (comme l'envoi ou la réception de messages) pour des objets précis, et de contrôler les opérations réflexives qui

leurs sont appliquées. Lorsqu'une opération réflexive est exécutée, le programme effectue un *saut* vers le méta-niveau. Ces *sauts* sont réifiés sous la forme de points de méta-contrôle, nommés *Reflectopoints*, qui sont des méta-objets créés spécifiquement pour chaque appel réflexif. Chaque *Reflectopoint* porte des méta-informations contextuelles au *saut* vers le méta-niveau, et contrôle l'exécution et le résultat des opérations réflexives demandées par le programme.

Par exemple, le *Reflectogram* permet de contrôler la réception de messages pour un receveur spécifique. Un message particulier peut-être filtré par une condition, et une action peut-être exécutée à la place ou en complément de la méthode qui correspond au message reçu. Le *Reflectopoint* associé au contrôle de la réception du message fournit les méta-informations contextuelles à l'opération réflexive (sélecteur du message, arguments...) et contrôle cette dernière. Ce contrôle peut consister en particulier à instrumenter le comportement de base du receveur, et modifier ou remplacer la réponse de celui-ci au message reçu, achevant ainsi une adaptation comportementale à granularité objet.

La variation comportementale, par l'utilisation du *MOP* fournit par le *Reflectogram*, est ici non-intrusive et reste strictement dans le paradigme objet. S'agissant de méta-contrôle, les objets pour lesquels des actions réflexives sont déclenchées ne sont pas modifiés. Leur identité est donc préservée tout au long de l'exécution du programme. Cependant, il n'y a pas de support explicite pour spécifier et retrouver des objets à instrumenter dans le programme en cours d'exécution. La propriété de minage d'objet n'est donc pas satisfaite. S'il semble possible de désactiver dynamiquement des instrumentations, Papoulias *et al.* n'explicitent pas comment annuler complètement le contrôle du comportement d'un objet par le *Reflectogram*. Nous considérons donc que la propriété de réversibilité n'est pas complètement satisfaite. Enfin, si le *Reflectogram* peut être utilisé pour instrumenter dynamiquement un programme, il s'agit d'un outil destiné à un-e utilisateur-rice final-e plutôt qu'à être utilisé comme base pour la mise en oeuvre d'autres concepts [Papoulias et al., 2017]. Nous considérons donc que la propriété de flexibilité n'est pas entièrement satisfaite.

Discussion. La table 2.4 résume le positionnement des approches réflexives étudiées dans les paragraphes précédents. L'utilisation de *MOPs* et l'association dynamique de méta-objets à des objets ou à leurs classes permettent de maintenir une séparation claire entre le code de base du programme (ou son code métier) et la variation comportementale. L'association et la dés-association d'un méta-objet et d'un objet ne nécessitent pas non plus de changer la structure de l'objet. Plutôt que remplacer l'objet (ou sa classe) par une autre version de lui-même (ou de sa classe), les *MOPs* modifient la sémantique de l'envoi et de la réception de messages (ou du *lookup*), et s'en servent pour contrôler le comportement à partir des méta-objets. Les propriétés de préservation d'identité et de non-intrusivité sont donc toujours

● Totalement supporté ◑ Partiellement supporté ○ Non supporté

	Granularité objet	Minage d'objets	Réversibilité	Identité préservée	Flexibilité	Non-intrusivité	Paradigme objet
MetaclassTalk	◑	○	●	●	○	●	●
Iguana/J	●	○	●	●	○	●	●
Talents	●	○	●	●	◑	●	●
Reflectivity	○	○	◑	●	◑	●	●
Reflectogram	●	○	◑	●	◑	●	●

TABLE 2.4 – Positionnement des approches réflexives basées sur les protocoles à méta-objets pour l'adaptation non-anticipée.

satisfaites par ces approches. Il manque cependant des abstractions de plus haut niveau pour définir et contrôler l'adaptation comportementale. Seuls les *Talents*, *Iguana/J* et le *Reflectogram* permettent de placer une sémantique sur un ensemble de modifications comportementales à appliquer à un objet. Mais dans le premier cas il est nécessaire d'écrire et de compiler ce comportement manuellement dans des *talents*, et dans le second cas *Iguana/J* force l'écriture de code d'instrumentation des mécaniques de transmission et de réception des messages pour mettre en œuvre l'adaptation. Dans le cas du *Reflectogram*, la flexibilité de la solution est limitée par sa conception en tant qu'API qui n'est pas destinée à être étendue.

2.3.3 Discussion

Les approches réflexives permettent d'adapter des objets de manière non-anticipée. Les difficultés majeures qui sont rencontrées se situent principalement au niveau de la flexibilité des solutions et des contraintes qui sont posées pour les utiliser − par exemple l'utilisation d'un langage particulier ou la nécessité de connaître des détails de mise en œuvre avancés du langage. Il y a également un manque général d'abstractions pour définir les instrumentations, ce qui ne permet pas de placer une sémantique claire sur l'adaptation et notamment en terme de contextes. Si les approches à base de *MOPs* permettent de séparer clairement l'instrumentation du programme de son comportement de base, il n'y a pas de contextualisation claire entre le code mettant en œuvre la variation comportementale et le code qui représente son comportement. Par exemple, une instrumentation conditionnelle du *lookup*

permet de décider entre plusieurs méthodes à exécuter, mais le lien entre la condition et le comportement n'est pas établi autrement que par l'instrumentation même du lookup. Plus que des finalités, les approches réflexives constituent des techniques et des mécaniques de base sur lesquelles il est possible de reposer pour la mise en œuvre de modèles dédiés à l'adaptation non-anticipée de comportement. À noter que l'aspect non-anticipé nécessite la présence d'un support pour être appliqué. Nombre de solutions réflexives reposent pour cela sur les environnements de *Live Programming*, mais le DSU pourrait tout à fait servir de support pour les langages ne supportant pas le *Live Programming*.

2.4 Infrastructures dédiées à l'adaptation

Il existe, à notre connaissance, peu de solutions complètes exclusivement dédiées à l'adaptation non-anticipée de comportement. Cette section décrit deux infrastructures notables au regard des problématiques étudiées, basées sur la réflexivité pour l'évolution et l'adaptation non-anticipée des programmes en cours d'exécution. Il s'agit de *Chisel*, une solution dédiée à l'adaptation non-anticipée, et de *LyRT*, une solution dédiée à la variation comportementale non-anticipée à base de rôles.

2.4.1 *Chisel*

Chisel [Keeney and Cahill, 2003, Keeney, 2004] est une infrastructure générique pour l'adaptation dynamique non-anticipée. Les objets et les classes peuvent être adaptés par association avec des méta-objets qui fournissent des variations comportementales sous forme de comportement fonctionnel et non-fonctionnel. Pour spécifier une adaptation comportementale, l'utilisateur-rice doit définir un nouveau méta-objet − dans le même langage que le programme en cours d'exécution − qui est chargé et compilé dynamiquement. L'utilisateur-rice doit ensuite demander l'association de ce méta-objet avec une classe ou un objet.

Les associations sont dynamiquement appliquées ou modifiées suivant des *politiques* définies par l'utilisateur-rice. Les *politiques* d'adaptation sont composées de règles qui spécifient les contraintes du déclenchement ou de l'arrêt de l'adaptation. Ces contraintes explicitent les évènements contextuels selon lesquels le comportement des objets du programme doit varier. Les *politiques* d'adaptation sont exprimées dans un langage de script dédié, spécifique à *Chisel*, et peuvent également être chargées et activées dans le programme en cours d'exécution. L'infrastructure fournit des *politiques* prédéfinies pour l'introspection du code en cours d'exécution et la mise à jour de comportements. Par exemple, une *politique* est prédéfinie pour adapter une classe et tracer son exécution pour observer et inspecter les objets qui constituent l'état de ses instances.

Chisel utilise un répertoire d'objets, qui référence les objets d'intérêt par un identifiant unique. Ces objets doivent être manuellement nommés, et nécessitent une intervention de l'utilisateur-rice pour observer le programme, identifier ces objets et les nommer. Les objets sont ensuite référencés par le répertoire, et disponibles pour être soumis à une variation comportementale. C'est une solution gourmande en ressource, car tous les objets doivent être sauvegardés dans une base de données pour être inspectés par l'utilisateur.

Chisel permet également de modéliser des contextes, sous la forme de *variables contextuelles*. Ces variables sont des objets qui sont définis et instanciés de manière statique − ils doivent être présents dans le programme avant son démarrage. Une *variable contextuelle* définit une condition et un évènement à soulever suivant le résultat de l'évaluation de la condition. Suivant l'évolution de l'état du programme, le contexte modélisé par les *variables contextuelles* change et le système est adapté au gré des évènements soulevés.

Enfin, *Chisel* fournit un outillage pour définir et visualiser des évènements et les *politiques* d'adaptation ou encore pour visualiser les traces d'inspection et les objets du répertoire d'objets.

L'utilisation de *Chisel* force à sortir partiellement du paradigme objet. L'utilisation d'un langage de script pour définir l'adaptation oblige à se construire une autre représentation mentale du programme alors même que le comportement des adaptations est défini dans le même langage que le programme. La flexibilité de l'architecture est réduite, car bien que son modèle soit dit *généraliste*, ce dernier se base sur *Iguana/J* pour construire ses méta-objets et comme mécanisme d'adaptation. L'infrastructure est donc limitée au langage Java, et aux machines virtuelles Java qui peuvent exécuter des programmes écrits avec *Iguana/J*.

2.4.2 *LyRT*

LyRT est une infrastructure basée sur les rôles, évaluée et brièvement décrite dans la section 2.2.2 [Taing et al., 2016a, Taing et al., 2016b, Taing et al., 2017, Taing, 2017]. *LyRT* est axé sur la variabilité et l'évolution dynamique du comportement d'un système en cours d'exécution. Son architecture repose sur un environnement d'exécution dédié à base de rôles, avec un support et un outillage pour l'adoption tardive de variations comportementales. Au delà des éléments déjà décrits dans le cadre des rôles, nous précisons ici quelques aspects de l'infrastructure de *LyRT* concernant l'adaptation non-anticipée.

Les variations comportementales sont modélisées par des rôles sous la forme de classes Java (*LyRT* est actuellement mis en œuvre avec le langage Java). Les rôles sont (re-)compilés puis chargés dans le programme en cours d'exécution. Leur application, en revanche, est spécifiée sous forme de documents *XML* persistants. Il faut dans ces scripts énumérer les objets de base, désignés par un identifiant

unique, puis les rôles qui doivent leur être liés ou déliés, ainsi qu'éventuellement une méthode à invoquer immédiatement après l'opération qui (dés-)associe le rôle. L'infrastructure détecte automatiquement les modifications apportées aux scripts *XML*, les interprète et applique les nouvelles adaptations. La spécification de l'application de l'adaptation se fait donc dans un formalisme différent de celui de sa sémantique.

La modification comportementale n'est appliquée que lorsque les entités concernées (objets, rôles) sont dans un état compatible avec l'adaptation. Un objet ne doit pas changer radicalement le comportement de ses méthodes si celles-ci sont en cours d'exécution, et qu'un ensemble de méthodes entre plusieurs entités doivent fournir un comportement cohérent. L'exemple fourni par Taing et al. illustre le problème sur un système de chiffrement de données, qui doit utiliser le même protocole de chiffrement pour un fichier en cours de transfert [Taing et al., 2016b]. Si le format de chiffrement change au milieu du transfert suite à une adaptation, les données transmises seront incohérentes. *LyRT* met en oeuvre un mécanisme de *Tranquilité* [Vandewoude et al., 2007], qui permet d'encapsuler certains comportements dans des transactions. Tant qu'une transaction est en cours d'exécution, les objets engagés dans cette transaction ne peuvent pas être adaptés. La sûreté de l'adaptation est une préoccupation critique pour maintenir le programme dans un état et un comportement cohérent. Nous considérons cet aspect comme orthogonal à la capacité d'adaptation, et il n'est donc pas étudié du point de vue de cette thèse.

2.4.3 Discussion.

La table 2.5 résume le positionnement des infrastructures dédiées à l'adaptation non-anticipée de comportement.

● Totalement supporté ◑ Partiellement supporté ○ Non supporté

	Granularité objet	Minage d'objets	Réversibilité	Identité préservée	Flexibilité	Non-intrusivité	Paradigme objet
Chisel	●	◑	●	●	○	●	◑
LyRT (rôles)	●	◑	●	●	◑	●	◑

Table 2.5 – Positionnement des infrastructures dédiées à l'adaptation non-anticipée.

Les limitations de *LyRT* ont déjà été discutées dans le cadre de la programmation par les rôles (section 2.2.2), et nous considérons qu'il s'agit de l'état de l'art de l'adaptation non-anticipée avec les rôles.

Les deux solutions nécessitent que le programme s'exécute en utilisant leur environnement d'exécution, et soit développé en utilisant leurs abstractions. Si un programme n'a pas été écrit avec *LyRT* ou *Chisel*, il n'y a pas de garanties de pouvoir injecter l'environnement et de pouvoir l'utiliser pleinement si le programme est déjà en cours d'exécution.

La sélection d'objets, bien que possible, ne passe pas l'échelle. Premièrement, les objets doivent *exister* pour être visibles dans des traces ou dans un outil afin d'être *marqués* par l'utilisateur-rice ou par l'environnement. Cela exclût par définition les objets temporaires et à durée de vie très courte. L'utilisateur-rice ne peut donc pas observer ces objets, les marquer ou les référencer dans les spécifications des adaptations. Secondement, la sélection d'objets pour les associer à des adaptations (méta-objets ou rôles) se fait manuellement. La question se pose de savoir comment, en pratique, trouver des objets d'intérêt dans des programmes qui font *vivre* simultanément des centaines ou des milliers d'instances d'une même classe. D'autre part, aucune des solutions ne propose de conditionner la sélection d'objets autrement que par la décision et l'action manuelle de l'utilisateur-rice. La propriété de minage d'objets n'est donc que partiellement satisfaite.

2.5 Synthèse et conclusion

● Totalement supporté ◑ Partiellement supporté ○ Non supporté

	Granularité objet	Minage d'objets	Réversibilité	Identité préservée	Flexibilité	Non-intrusivité	Paradigme objet
Adaptation avec minage d'objets							
Adaptation basée sur les groupes (COP)	●	◑	●	●	◑	◑	○
Chisel (Iguana/J)	●	◑	●	●	○	●	◑
LyRT (rôles)	●	◑	●	●	◑	●	◑

TABLE 2.6 – Positionnement des approches étudiées pour l'adaptation non-anticipée.

L'évaluation des solutions étudiées pour l'adaptation non-anticipée de comportement dans les programmes à objets est synthétisée dans la table 2.7. Toutes les propriétés sont globalement atteignables, sauf le minage d'objets, mais jamais au sein de la même solution ou seulement de manière partielle. Il est notamment difficile de réunir certaines propriétés selon la catégorie de solutions étudiées. Par exemple, les paradigmes de programmation que sont l'AOP, COP ou la programmation par les rôles, peinent à rester dans le paradigme objet mais permettent une granularité fine et offrent la possibilité d'annuler les adaptations. Les approches réflexives sont non-intrusives et restent dans le paradigme objet, mais sont peu flexibles, ne fournissent pas d'abstraction pour contrôler l'adaptation − notamment pour la propriété de réversibilité − et ne permettent pas toujours une granularité objet. Enfin, la propriété de minage d'objets n'est jamais complètement atteinte, et uniquement pour un nombre très restreint de solutions. Une seconde synthèse, avec une mise au point sur ces approches, est présentée par la table 2.6.

Le minage d'objet est partiellement possible avec ces trois solutions, qui sont respectivement l'adaptation basée sur les groupes, liée à la programmation orientée contexte, l'infrastructure *Chisel* basée sur la réflexivité et l'infrastructure *LyRT* basée sur les rôles. Les aspects du minage d'objets ne sont cependant pas les mêmes. L'adaptation basée sur les groupes permet plus de souplesse et de finesse dans la spécification des groupes d'objets, mais cette souplesse et cette finesse varient selon les mécanismes de sélection d'objets. La spécification des groupes d'objets doit être anticipée, ou alors être introduite dans le code du programme. *Chisel* et *LyRT* permettent de sélectionner des objets à adapter de manière complètement non-anticipée et non-intrusive, mais n'ont aucune expressivité dans cette sélection qui doit être effectuée manuellement par l'utilisateur-rice.

Bien que ces approches semblent prometteuses par rapport aux autres solutions au regard de leur évaluation, spécifiquement en ce qui concerne le minage d'objets, elles sont limitées par le fait qu'il s'agit de solutions destinées à un-e utilisateur-rice final-e. Leur flexibilité est limitée, notamment par l'utilisation de langages ou de constructions de langage particulières ou par des contraintes de portabilité et de mise en œuvre. De plus, ces solutions étant *terminales* et reposant sur des modèles et des environnements d'exécution qui leurs sont propres, il n'est pas possible de les combiner pour obtenir la satisfaction de toutes les propriétés recherchées. Elles ont cependant toutes comme point commun de reposer sur la réflexivité comportementale, qui bien que non-apparente sert à la mise en œuvre des différents modèles qui composent ces solutions. Les approches réflexives représentent alors le meilleur point de départ pour considérer toutes les propriétés que nous recherchons pour l'adaptation non-anticipée de comportement dans les programmes à objet. Les chapitres suivants décrivent notre approche pour satisfaire les manques de l'état de l'art au regard de notre évaluation, ainsi que sa mise en œuvre à partir de mécanismes de réflexivité comportementale.

● Totalement supporté ◑ Partiellement supporté ○ Non supporté

	Granularité objet	Minage d'objets	Réversibilité	Identité préservée	Flexibilité	Non-intrusivité	Paradigme objet
Mise à jour et reconfiguration							
DSU	○	○	●	◑	●	○	●
Live Programming	○	○	●	◑	●	○	●
Prog. orientée composants	○	○	●	◑	◑	◑	○
Paradigmes pour l'adaptation							
Programmation Orientée Aspects	●	○	●	●	●	●	○
Programmation par les Rôles	●	◑	●	●	◑	●	◑
⋆Programmation Orientée Contexte	●	◑	●	●	◑	◑	○
Approches réflexives							
Sous-classes anonymes	●	○	●	●	◑	●	●
Method Wrappers	○	○	◑	●	●	●	●
Wrappers + Sous-classes anonymes	●	○	◑	●	◑	●	●
Techniques à base de proxy	●	○	●	○	○	●	●
MetaclassTalk	◑	○	●	●	○	●	●
Iguana/J	●	○	●	●	○	●	●
Talents	●	○	●	●	◑	●	●
Reflectivity	○	○	◑	●	◑	●	●
Reflectogram	●	○	◑	●	◑	●	●
Infrastructures dédiées							
Chisel	●	◑	●	●	○	●	◑
LyRT (rôles)	●	◑	●	●	◑	●	◑

⋆ Avec les mécanismes d'adaptation basée sur les groupes.

TABLE 2.7 – Positionnement des approches étudiées pour l'adaptation non-anticipée.

Deuxième partie

Patron de langage objet dynamique pour l'adaptation non-anticipée de comportement

Cette partie contient un seul chapitre, qui présente la proposition centrale de la thèse. Il s'agit d'un patron de langage, *Kernel-Lub*, qui représente un modèle minimal de langage objet dynamiquement typé. Les langages conformes à ce modèle minimal peuvent être étendus par *Lub* et par les *Collecteurs*. Ce sont des patrons extensions de langage, qui introduisent respectivement des capacités d'adaptation non-anticipée à granularité objet et la possibilité de spécifier de manière non-anticipée les groupes d'objets qui seront soumis à l'adaptation.

Chapitre 3

Patron de langage objet pour l'adaptation non-anticipée de comportement

3.1 Kernel-Lub : un langage objet minimal 73
3.1.1 Infrastructure de Kernel-Lub 73
3.1.2 Opérations du langage 74
3.1.3 Modèle d'exécution 75
3.1.4 Sémantique opérationnelle 76
3.2 Lub : une extension pour l'adaptation non-anticipée de comportement 77
3.2.1 Infrastructure de Lub 78
3.2.2 Opérations du langage 78
3.2.3 Modèle d'exécution 79
3.2.4 Sémantique opérationnelle 80
3.2.5 Adaptations gardées 81
3.2.6 Conception d'adaptation comportementale avec Lub 82
3.2.7 Impact de l'adaptation sur le lookup 83
3.3 Collecteurs : groupes d'objets dynamiques pour l'adaptation collective 87
3.3.1 Infrastructure des Collecteurs 87
3.3.2 Opérations du langage 88
3.3.3 Modèle d'exécution 89
3.3.4 Sémantique opérationnelle 89
3.3.5 La collecte dynamique d'objets : une illustration 92
3.4 Propriétés fondamentales 94
3.4.1 Kernel-Lub : formalisation du lookup 94
3.4.2 Lub : formalisation du lookup 98
3.5 Conclusion 102

Dans cette thèse, nous nous intéressons au déverminage non-anticipé de programmes en cours d'exécution. Nous avons à ce titre introduit dans le chapitre 1 l'état de la pratique du déverminage et pointé ses limites du point de vue de cette problématique. Nous avons alors proposé d'utiliser des techniques d'adaptation non-anticipée de comportement pour pouvoir déboguer un programme de manière imprévue. Ce type d'adaptation consiste à instrumenter les objets du programme pour faire varier leur comportement. L'adaptation non-anticipée est notamment utilisée pour observer et modifier un programme alors que cela n'était pas prévu avant son exécution. Nous avons donc établi sept propriétés désirables pour l'adaptation d'objets dans un programme en cours d'exécution et dressé un état de l'art de l'adaptation non-anticipée dans le chapitre 2, que nous avons évalué au regard de ces propriétés. Aucune des solutions étudiées ne permet de satisfaire pleinement toutes les contraintes posées pour aborder notre problématique.

Dans ce chapitre, nous présentons notre approche pour satisfaire ces propriétés, et déverminer des programmes en cours d'exécution par le biais de l'adaptation non-anticipée. Nous proposons d'étendre le langage utilisé pour développer le programme avec un support complet pour l'adaptation du comportement des objets. Notre solution est présentée sous la forme d'un patron de langage, qui permet d'étendre le modèle des langages objets dynamiquement typés par une extension légère, rendant possible l'adaptation non-anticipée à granularité objet.

Nous présentons d'abord un modèle de langage hôte, nommé *Kernel-Lub*, plus petit dénominateur commun de ces langages objets dynamiques. Pour être éligible à une extension par notre patron de langage, le modèle minimal d'un langage doit être conforme à *Kernel-Lub*. Le patron, basé sur *Kernel-Lub*, est composé de deux parties indépendantes. La première, nommée Lub, active la capacité d'adapter dynamiquement le comportement d'objets spécifiques du programme en cours d'exécution. La seconde partie du patron, nommée Collecteurs, permet de retrouver et de regrouper dynamiquement des objets puis de les rendre accessible au reste du programme au travers d'une interface. La combinaison des deux parties du patron permet l'adaptation de groupes d'objets récupérés dynamiquement dans le programme en cours d'exécution. Ces extensions de langage sont légères d'un point de vue modèle, mais déverrouillent les capacités nécessaires pour aborder les contraintes posées pour le déverminage d'objets par l'adaptation non-anticipée.

Nous présentons d'abord le modèle de chaque partie du patron, c'est-à-dire *Kernel-Lub*, Lub, puis les Collecteurs. Chacune de ces parties est ensuite illustrée au travers d'exemples plus concrets, qui décrivent sur des cas d'utilisation simples l'impact et l'utilisation du patron de langage. Le cadre formel du patron de langage est décrit en fin de chapitre.

3.1 Kernel-Lub : un langage objet minimal

Kernel-Lub est un modèle de langage minimal qui représente le plus petit dénominateur commun aux langages objets dynamiques destinés à être étendus avec des capacités d'adaptation. Ce langage minimal est décrit de telle manière qu'il pourrait être complété pour modéliser un langage existant, mais ne représente pas un langage opérationnel destiné à être utilisé pour programmer. Nous présentons l'infrastructure et les opérations de *Kernel-Lub*, qui représentent le méta-modèle du langage, c'est à dire les structures manipulées par un-e développeur-euse. Nous présentons ensuite le modèle d'exécution du langage, qui est la structure manipulée par l'environnement d'exécution du programme.

3.1.1 Infrastructure de Kernel-Lub

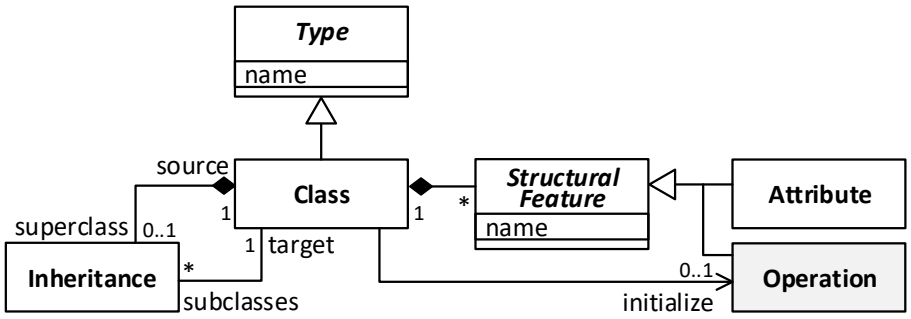


FIGURE 3.1 – Infrastructure de *Kernel-Lub*

L'infrastructure de *Kernel-Lub* est décrite dans la Figure 3.1. C'est un *patron* d'infrastructure minimale de langage objet dynamiquement typé.

Ce langage repose sur les notions de type et de classe. Un **Type** est une construction primitive du langage. Il s'agit d'un élément nommé. Une **Classe** est un type de base. Une classe est composée d'une liste éventuellement vide d'attributs et d'opérations. Ce sont des éléments structurels nommés (*StructuralFeature*). Une des opérations peut éventuellement être dédiée à l'initialisation des instances d'une classe (*initialize*). Pour une classe A donnée, l'opération désignée par *initialize* doit être une opération de la classe A. Le langage est à héritage simple, c'est-à-dire qu'une classe ne peut hériter au plus que d'une seule autre classe.

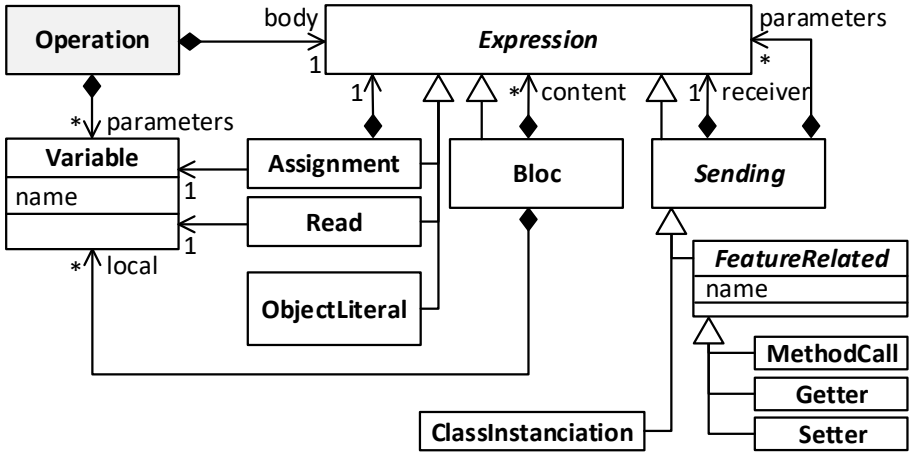


FIGURE 3.2 – Opérations du langage minimal

3.1.2 Opérations du langage

La Figure 3.2 représente le *patron* d'opérations et d'expressions qui complète le *patron* d'infrastructure de langage de la Figure 3.1.

Opération. Une opération est constituée d'une série de paramètres formels non typés (des variables) et d'un corps (une expression).

Expression. Les expressions sont évaluées à l'exécution du programme et produisent une valeur primitive ou une référence d'objet (section 3.1.3). Une expression est un concept abstrait qui se décline dans les catégories concrètes suivantes :

Assignment : Affectation du résultat de l'évaluation d'une expression quelconque à une variable.

Read : Lecture du contenu d'une variable.

ObjectLiteral : En tant qu'expressions, les littéraux d'objets désignent des instances de littéraux prédéfinies. Voir *Littéraux d'objets* ci-dessous.

Bloc : Un bloc est une expression comportant une liste de variables locales et une liste de sous-expressions évaluées en séquence.

Sending : Envoi d'un message à un objet receveur, correspondant à la valeur de l'expression *receiver* associée. Une série éventuellement vide de paramètres effectifs, désignée par *parameters*, est associée à l'envoi de message.

Littéraux d'objets. La classe *ObjectLiteral* représente des *références* prédéfinies d'objets, dont les valeurs numériques, les valeurs booléennes, les caractères, les

chaînes de caractères, ainsi que les références d'objets spéciales suivantes :

self : Représente le receveur d'une opération dans l'expression que constitue son corps (*body* dans la Figure 3.2).

super : Alias de *self* utilisable en position de receveur d'un appel de méthode, c'est-à-dire l'expression jouant le rôle *receiver*, associée à la classe *MethodCall* dans la Figure 3.2. L'utilisation de *super* permet de démarrer le *lookup* de la méthode appelée dans la super-classe du receveur (section 3.1.3).

null : Représente une référence d'objet non valide, dite *nulle*.

Envoi de message. On distingue deux catégories d'envoi de messages : les messages liés à un champ du receveur (*FeatureRelated*) et les autres. Pour *Kernel-Lub*, ces derniers sont réduits à la seule opération d'instanciation de classe.

FeatureRelated Le champ du receveur concerné par le message est désigné par son nom (attribut *name* de *FeatureRelated*). Le cas général d'envoi de message est l'appel de méthode (*MethodCall*), qui consiste à envoyer au receveur un message recevable conformément à son interface. Dans les cas particuliers de *Getter* et *Setter*, l'attribut *name* désigne un attribut du receveur dont on souhaite lire le contenu (*Getter*), ou que l'on souhaite mettre à jour (*Setter*).

Instanciation L'instanciation de classe (*ClassInstantiation*) est un cas particulier d'envoi de message *contraint*. Le receveur est un méta-objet qui désigne la classe à instancier. Les éventuels paramètres sont transmis à la méthode d'initialisation de la classe à instancier. Cette méthode doit correspondre à l'opération désignée par *initialize*.

3.1.3 Modèle d'exécution

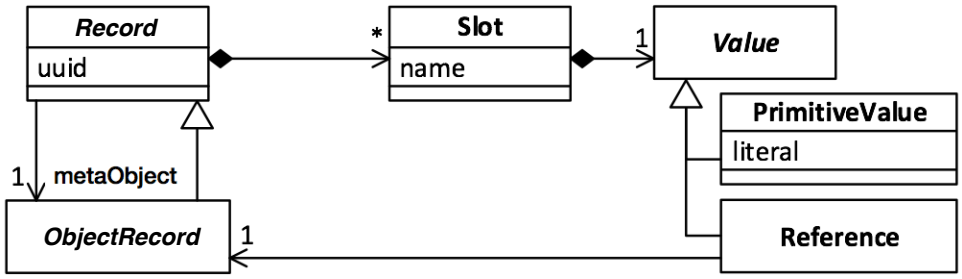


FIGURE 3.3 – Modèle d'exécution minimal

Le modèle de l'environnement d'exécution d'un programme *Kernel-Lub* est représenté par la Figure 3.3. Les éléments du modèle d'exécution sont décrits ci-après.

Record. *Record* est une classe abstraite qui représente un objet en mémoire. Un record est désigné par un unique identifiant *uuid* (alias de référence mémoire). Il rassemble des cases mémoire nommées (*slots*). Ces cases désignent soit les attributs de l'objet, soit les variables temporairement associées à l'objet.

ObjectRecord. L'unique sous-classe concrète de *Record* est *ObjectRecord*, dont les instances sont les objets du programme. Une référence désigne une unique instance d'*ObjectRecord*, grâce à la résolution d'*uuid*. Chaque instance d'*ObjectRecord* est toujours associée à exactement une instance d'*ObjectRecord*, qui joue le rôle d'instance de méta-classe (*metaObject*). Les classes sont également des instances de *ObjectRecord*, c'est à dire des objets.

Slot. Un *slot* stocke une valeur primitive ou la référence d'un objet correspondant à un *record*. Un *slot* d'attribut stocke un attribut direct ou hérité de l'objet auquel il est associé. Un *slot* d'attribut est associé à un objet pendant toute la durée de vie de ce dernier. Un *slot* de paramètre stocke un paramètre de méthode. Un *slot* de paramètre est associé à l'appel d'une méthode, lui-même associé à un receveur. Le receveur correspond au *record* associé au *slot*. Un *slot* de paramètre n'est associé à un *record* que pendant l'exécution de la méthode.

3.1.4 Sémantique opérationnelle

La sémantique opérationnelle du langage repose sur le modèle d'exécution décrit par la Figure 3.3, qui sert de cadre à la sémantique d'exécution des opérations du langage (Figure 3.2). La notion de *lookup*, évoquée dans cette section, est formellement définie dans la section 3.4.1.

Instanciation. À l'exécution de $ClassInstanciation$, une nouvelle instance $\boldsymbol{i}$ de *ObjectRecord* est créée avec un nouvel *uuid*. Tous les *slots* d'attributs sont créés conformément à la spécification de la classe instanciée. L'instance de *ObjectRecord* qui représente la classe de l'instance $\boldsymbol{i}$ est créée si elle n'existe pas déjà. Elle est associée à la nouvelle instance $\boldsymbol{i}$ précédemment créée, et joue le rôle de méta-objet pour $\boldsymbol{i}$. La méthode *initialize* de la classe est exécutée pour $\boldsymbol{i}$, si elle existe.

Envoi de message lié à un champ. À l'exécution de *Getter* ou de *Setter*, le champ désigné correspond à un *slot* du receveur. Dans le cas de *Setter*, l'expression

associée est évaluée. Son résultat, une valeur primitive ou une référence, est stocké dans le *slot*. Dans le cas de *Getter*, le contenu du *slot* est renvoyé en résultat.

À l'exécution de *MethodCall*, le champ désigné correspond à une méthode. Dans le cas général, le *lookup* pour la méthode demandée démarre dans la classe représentée par le méta-objet associé au receveur de l'opération. Dans le cas particulier du receveur désigné par *super*, le *lookup* démarre dans la super-classe de l'objet pour lequel la méthode contenant l'opérateur *MethodCall* est en cours d'exécution.

Le *lookup* peut réussir ou échouer. Quand il réussit, le corps de la méthode obtenue est exécuté.

Lecture et affectation de variable. Une variable correspond à un *slot* associé à un objet. Cette association est temporaire dans le cas d'un paramètre de méthode. Dans le cas de l'affectation, l'expression associée est évaluée. Son résultat, une valeur primitive ou une référence, est stocké dans le *slot*. Dans le cas de la lecture, le contenu du *slot* est renvoyé en résultat.

Littéral d'objet. L'objet désigné est créé s'il n'a pas déjà été utilisé, comme décrit ci-dessus pour l'instanciation.

Bloc. À l'exécution d'un bloc de code, une nouvelle instance de *ObjectRecord* est créée. À chaque variable locale de ce bloc correspond un *slot* de l'instance de *ObjectRecord* qui représente le bloc. Toutes les sous-expressions du corps du bloc sont ensuite évaluées en séquence dans l'ordre de leur déclaration.

3.2 Lub : une extension pour l'adaptation non-anticipée de comportement

Lub permet l'adaptation non-anticipée de programmes lors de leur exécution, sous la forme d'une adaptation légère des langages de programmation et de leurs environnements. Lub introduit une seule construction de langage et un seul opérateur, puis repose sur les mécanismes présents du langage hôte pour activer l'adaptation dynamique. Lub est dynamiquement typé et ne supporte que l'héritage simple. Certains langages dynamiques, comme Smalltalk ou Python, fournissent une structure de base compatible avec une telle extension. Lub est conçu comme une abstraction de ce type de langage, et nous le présentons alors comme un *patron de langage* pour l'adaptation dynamique dans les langages orientés objet de la même classe que *Kernel-Lub*. Lub est donc construit par extension du *patron* de langage *Kernel-Lub*.

3.2.1 Infrastructure de Lub

La Figure 3.4 représente le *patron* d'infrastructure de langage de la Figure 3.1 étendu par le concept d'*Adaptation*.

La spécificité de Lub repose sur la notion d'*Adaptation*. C'est un nouveau type du langage. Il s'agit d'un élément nommé qui peut être instancié. Une adaptation est associée à une classe existante, et sélectionne un sous-ensemble de ses attributs et/ou de ses opérations. Les champs ciblés par l'adaptation sont nécessairement des champs de la classe associée à l'adaptation. Concrètement, *subset* est un sous-ensemble des $StructuralFeatures$ associées à $target$.

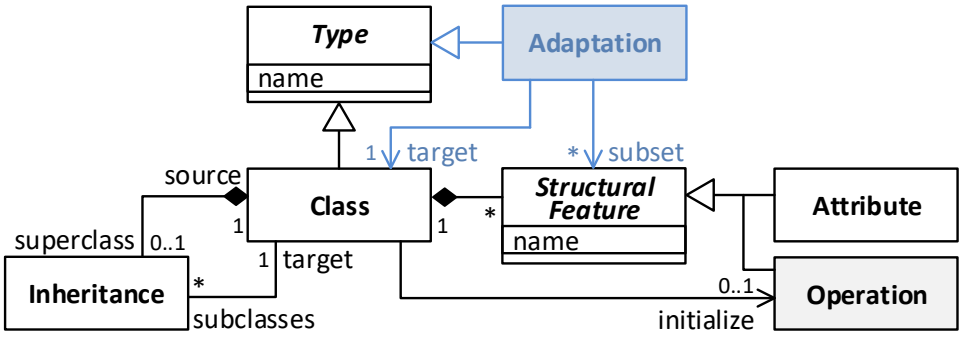


FIGURE 3.4 – Infrastructure de Lub : *Kernel-Lub* étendu avec Adaptation

3.2.2 Opérations du langage

La Figure 3.5 représente le *pattern* d'opérations et d'expressions de la Figure 3.2 étendu par le concept d'*Adaptation*.

Un nouveau type d'envoi de message est rajouté au modèle initial : **ObjectAdaptation**. Il s'agit comme pour l'instanciation de classe (*ClassInstanciation*) d'un cas particulier d'envoi de message *contraint*. Le receveur est un objet à adapter. L'unique paramètre de *ObjectAdaptation* est une expression dont l'évaluation doit produire une instance de la classe *Adaptation*, c'est-à-dire un méta-objet qui désigne l'adaptation à appliquer au receveur.

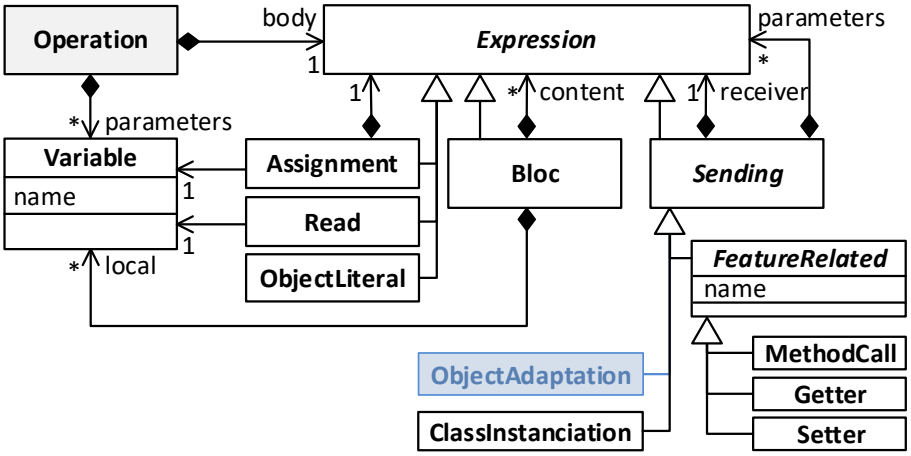


FIGURE 3.5 – Opérateurs de Lub étendus avec Adaptation

3.2.3 Modèle d'exécution

La Figure 3.6 représente le modèle d'environnement d'exécution de la Figure 3.3 étendu par le concept d'*Adaptation*.

LubRecord. Dans Lub, le type *Record* est désormais décliné en *ObjectRecord* et en *LubRecord*. Une adaptation, c'est-à-dire une instance de *Adaptation* (Figure 3.4), réfère à une instance de la classe *LUBRecord*. Un *LubRecord* sert avant tout à désigner la classe de démarrage du *lookup* : c'est désormais un *LubRecord* qui joue le rôle de méta-objet pour les instances de *ObjectRecord*.

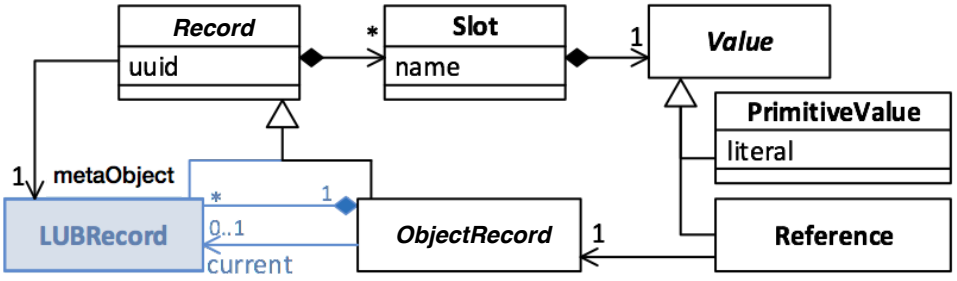


FIGURE 3.6 – Modèle d'exécution de Lub étendu avec Adaptation

ObjectRecord. Un objet standard est représenté par un *ObjectRecord*. Il peut être complété par un ensemble éventuellement vide de *LubRecord*. Chaque *LubRecord* correspond à une adaptation issue de l'exécution de *ObjectAdaptation*. Une instance de *ObjectRecord* peut alors être *adaptée*, et dans ce cas une instance de *LUBRecord*, c'est-à-dire une adaptation, lui est associée. À un instant donné, au plus une adaptation s'applique : il s'agit de l'adaptation nommée *current* dans la Figure 3.6. Les autres instances de *LubRecord* accessibles depuis un *ObjectRecord* permettent de mettre en œuvre des transitions entre adaptations. Par exemple : adaptation par x, puis par y, puis retour à l'adaptation par x.

3.2.4 Sémantique opérationnelle

La sémantique opérationnelle de Kernel Lub est étendue au concept d'*adaptation*. Le rôle d'instance de méta-classe est alors joué par une instance de *LUBRecord* au lieu de *ObjectRecord*. Par construction, les méta-objets ne peuvent donc pas être adaptés. L'instance de méta-classe définit toujours, comme dans le langage de base, le point de départ du lookup des méthodes.

Création d'adaptation. À l'exécution de *ObjectAdaptation*, une nouvelle instance $\boldsymbol{l}$ de *LUBRecord* est créée avec un nouvel *uuid* si elle n'existe pas déjà pour l'objet receveur. Tous les *slots* sont créés conformément à la classe ciblée par l'adaptation (*target*) et au filtre associé (*subset*). L'instance de *LUBRecord* qui représente la classe ciblée par l'adaptation est créée si elle n'existe pas déjà. Elle est associée à la nouvelle instance $\boldsymbol{l}$ de *LUBRecord* et devient son méta-objet.

Adaptation d'une instance. L'instance de *ObjectRecord* qui correspond à l'objet receveur de *ObjectAdaptation* est ensuite adaptée. La nouvelle instance $\boldsymbol{l}$ de *LUBRecord* la complète par composition et devient sa nouvelle adaptation *current*. Lorsqu'une instance de *ObjectRecord* est adaptée, seule cette instance particulière est affectée. L'instance courante de *LUBRecord* qui définit un nouveau point de départ du lookup n'affecte pas les autres instances de la classe *ObjectRecord*.

Suppression de l'adaptation d'une instance. L'adaptation courante est supprimée lorsque le littéral *null* est fourni en paramètre à *ObjectAdaptation*. L'instance *current* devient *null*, et le receveur retrouve son comportement d'origine.

Attributs adaptés. Si l'adaptation comporte un attribut x, c'est-à-dire que le filtre *subset* de l'adaptation comporte une instance de *Attribute* dont *name* correspond à x, alors l'instance de *LUBRecord* correspondant à cette adaptation comporte elle-même une instance de *Slot* dont *name* correspond à x. Deux cas

peuvent alors se présenter :

— x n'est pas un attribut de l'objet adapté : dans ce cas, le slot x est équivalent à un slot ajouté à l'objet adapté le temps de son adaptation

— x est un attribut de l'objet adapté : dans ce cas le slot x est équivalent à une nouvelle version du slot x, qui masque le slot x d'origine pendant la durée l'adaptation

Appel de méthode. Un objet peut être non adapté (cas nominal) ou adapté si une adaptation – c'est-à-dire une instance de *LubRecord* – lui a été associée.

— Si l'objet n'est pas adapté, l'instance de *ObjectRecord* qui le représente n'est pas associée à un *LUBRecord* représentant une adaptation (*current* est *null*). À l'exécution de *MethodCall*, le *lookup* de la méthode désignée démarre dans la classe représentée par le méta-objet associé au receveur de l'opération, ou dans sa super-classe en cas d'appel via *super*. Il s'agit du *lookup* standard du langage.

— Si l'objet est adapté, à l'exécution de *MethodCall* le *lookup* de la méthode désignée démarre dans la classe représentée par le méta-objet associé au *LUBRecord* qui joue le rôle de *current*. Le *lookup* peut éventuellement démarrer dans sa super-classe en cas d'appel via *super*. Si le *lookup* réussit, le corps de la méthode obtenue est exécuté. Sinon, le *lookup* continue à partir de la classe du receveur comme s'il n'était pas adapté.

3.2.5 Adaptations gardées

Une adaptation gardée – *GuardedAdaptation* – est une adaptation munie d'une expression qui joue le rôle de *garde*. C'est un sous type de *Adaptation*. La Figure 3.7 représente le *patron* d'infrastructure de langage de la Figure 3.1 étendu au concept d'*adaptation gardée*.

L'adaptation gardée permet de contextualiser l'adaptation, en conditionnant son activation et sa désactivation. L'ajout du concept d'adaptation gardée n'a pas d'incidence sur le modèle d'opérations et d'expressions de la Figure 3.2. Concrètement, lorsqu'une adaptation est programmée par appel de *ObjectAdaptation*, l'adaptation fournie en paramètre peut être complétée par une expression qui joue le rôle de *garde*. L'ajout du concept d'adaptation gardée n'a pas d'incidence sur le modèle d'exécution décrit par la Figure 3.6.

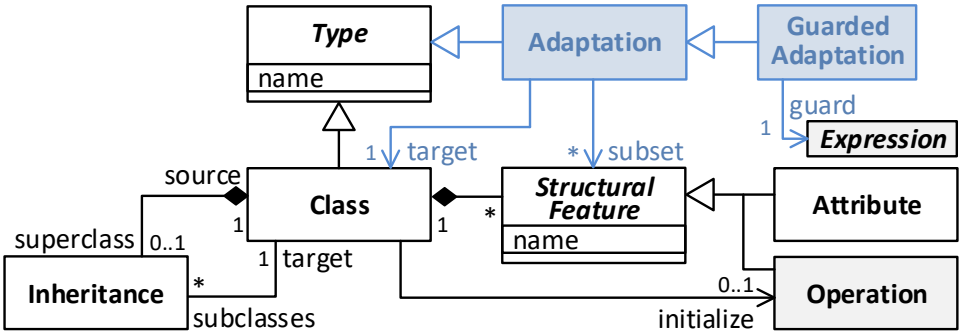


FIGURE 3.7 – Lub : infrastructure pour adaptations gardées

3.2.5.1 Sémantique Opérationnelle

La sémantique opérationnelle de Lub étendue au concept d'adaptation gardée est obtenue par traduction vers la sémantique opérationnelle de Lub étendue au concept d'adaptation de la section 3.2.4.

Appel de méthode. Si un objet est adapté par une adaptation gardée, à l'exécution de *MethodCall*, la garde associée est évaluée. Si le résultat correspond à la référence du littéral *true*, le *lookup* adapté démarre comme explicité dans le paragraphe *Appel de méthode* de la section 3.2.4. Sinon, l'adaptation est ignorée et le *lookup* démarre à partir de la classe du receveur comme s'il n'était pas adapté.

3.2.6 Conception d'adaptation comportementale avec Lub

Lub étant un langage objet minimal avec des capacités d'adaptation dynamiques, toute définition avec le langage reste dans le paradigme du langage hôte. Nous précisons les modalités de définition d'adaptation comportementale et de la composition de comportement avec un langage objet étendu par Lub.

3.2.6.1 Définir une modification comportementale

Une adaptation décrit, *in fine*, une modification du comportement d'un objet. Ce n'est que lorsque cette dernière est appliquée à l'exécution (Figure 3.6) que le comportement de l'objet est effectivement altéré. L'instanciation d'une adaptation nécessite de cibler une classe cible ainsi que les fonctionnalités qui seront adaptées (Figure 3.4). La variation comportementale est décrite et mise en œuvre dans la classe ciblée par l'adaptation.

Pour définir une modification de comportement, il faut donc soit créer une nouvelle classe avec ses attributs et ses opérations, soit réutiliser une classe pré-existante dans le programme en cours d'adaptation. Le développement du comportement altéré se fait donc sans changer de paradigme, puisqu'il s'agit simplement de créer de nouvelles classes tout à fait standards au regard du langage. La compatibilité avec les autres constructions du langage est maintenue, et il n'est pas nécessaire d'utiliser des constructions ou des concepts particuliers pour concevoir des adaptations.

3.2.6.2 Composition d'adaptations

Une adaptation de comportement peut être composée de différents comportements. Lors de l'adaptation, le comportement C d'un objet est remplacé par le comportement C' défini par la classe cible de l'adaptation. Le comportement C' peut être conçu comme un remplacement intégral de C, composé de différentes unités de comportement. Il peut aussi ajouter de simples préfixe et/ou suffixe comportementaux à C. Il s'agit d'une composition de comportements, à partir de diverses unités de comportement.

Lub ne permet pas la composition d'adaptations à proprement parler. En revanche, la composition de comportements dans une classe est explicitement possible par les mécanismes du langage. Pour Lub, il s'agit des mécanismes présents naturellement dans un langage objet, tels que l'héritage ou la délégation. Un autre langage hôte que *Kernel-Lub*, augmenté par Lub pour obtenir des capacités d'adaptation dynamiques, pourrait fournir d'autres mécanismes de composition plus poussés et plus souples. Dans tous les cas, les conflits potentiels inhérents à la composition de classes doivent être résolus au niveau du langage lui-même, et non pas au niveau de l'adaptation. C'est un problème lié, dans ce cas précis, exclusivement à la conception du comportement. La composition de comportements est donc indépendante du mécanisme d'adaptation. Dans Lub, la composition de comportements est donc implicite.

3.2.7 Impact de l'adaptation sur le lookup

La Figure 3.8 présente deux exemples d'adaptation d'un objet, et illustre l'effet de l'adaptation sur le lookup du langage minimal. Le modèle de la Figure 3.8 comprend une classe X, avec deux méthodes (*m1*, *m4*), ainsi qu'une classe J avec deux méthodes (*m1*, *m2*) et une classe K héritant de J avec une méthode (*m3*). Ce modèle peut être représenté à l'exécution par un modèle conforme à la Figure 3.6. Une instance de X est représentée par une instance de *ObjectRecord*, associée à une instance de *LUBRecord* qui joue le rôle de méta-objet (la classe X). Pour illustrer l'impact de l'adaptation sur le *lookup*, un objet x instance de X sera successivement modifié par des adaptations basées sur les classes J et K.

Un exemple d'adaptation d'un objet

Lorsque l'objet x reçoit un message, le lookup démarre à partir de son méta-objet, c'est à dire sa classe. Dans le cas de figure *(a)*, le lookup pour les méthodes *m1, m2, m3, m4* est démarré à partir de la classe X. Les méthodes qui ne font pas partie des opérations de X provoquent un échec du lookup, et donc une erreur à l'exécution. Dans le cas de figure *(b)*, x a été associé à une instance de *AdaptJ*. *AdaptJ* est une adaptation de X à partir de la classe J, et spécifie un comportement adapté pour la méthode *m1* de X ainsi qu'un nouveau comportement via une nouvelle méthode *m2*. Dans le modèle d'exécution, l'instance initiale de *ObjectRecord* (*x)* est désormais associée à un nouveau *LUBRecord* (*adaptJ*) qui joue le rôle de l'adaptation courante de l'objet. Le lookup pour les méthodes *m1, m2* est alors démarré à partir de ce *LUBRecord*, c'est-à-dire de la classe *J*. Le lookup pour les méthodes *m3, m4* est lui démarré à partir du méta-objet d'origine de x, c'est-à-dire la classe X, car elles ne font pas partie des opérations sélectionnées par l'adaptation *AdaptJ*. Dans le cas de figure *(c)*, x a été associé à une instance de *AdaptK* qui spécifie un nouveau comportement pour *m1* et *m3*. L'instance *adaptK* se substitue alors à *adaptJ* et devient le nouveau *LUBRecord* courant de x. Le lookup pour les méthodes *m1, m3* sera désormais démarré dans K tandis que le lookup pour *m2, m4* sera démarré dans le méta-objet original de x, c'est-à-dire la classe X.

Dans cette illustration, l'objet x est adapté par deux adaptations successives, *AdaptJ* et *AdaptK*. Ces adaptations sont des instances de *Adaptation*, la construction spécifique introduite par Lub. C'est un classificateur associé à une classe déjà présente dans le système. Ces adaptations spécifient un sous-ensemble des attributs et des méthodes de la classe associée, qui vont représenter la variation comportementale pour l'objet ciblé. Ici, *AdaptJ* et *AdaptK* spécifient l'adaptation d'une partie de l'interface de X, c'est-à-dire la méthode *m1*, ainsi que l'acquisition pour les instances de X de nouvelles méthodes (*m2, m3*). Il n'y a pas de contrainte sur le moment de l'adaptation : celle-ci peut-être conçue et appliquée à l'exécution. Les classes J et K peuvent donc être créées et leurs adaptations spécifiées alors que le programme est déjà en train de s'exécuter. En revanche une seule adaptation peut être associée à x à un moment donné. Si l'adaptation *adaptJ* est l'adaptation courante de x, le fait d'associer dynamiquement *adaptK* à x annule l'effet de *adaptJ* et *adaptK* devient la nouvelle adaptation courante.

Résultats du lookup

Les résultats du lookup suivant les scénarios *(a)*, *(b)* et *(c)* de la Figure 3.8 sont détaillés dans la Table 3.1. Pour chaque méthode, le nom de la classe dans laquelle le lookup a été résolu est imprimé, suivi du nom de la méthode. Lorsque le lookup échoue, le message *ERREUR* est inscrit dans la table. Pour chaque scénario, les messages *m1, m2, m3, m4* sont envoyés successivement à l'objet x instance de X.

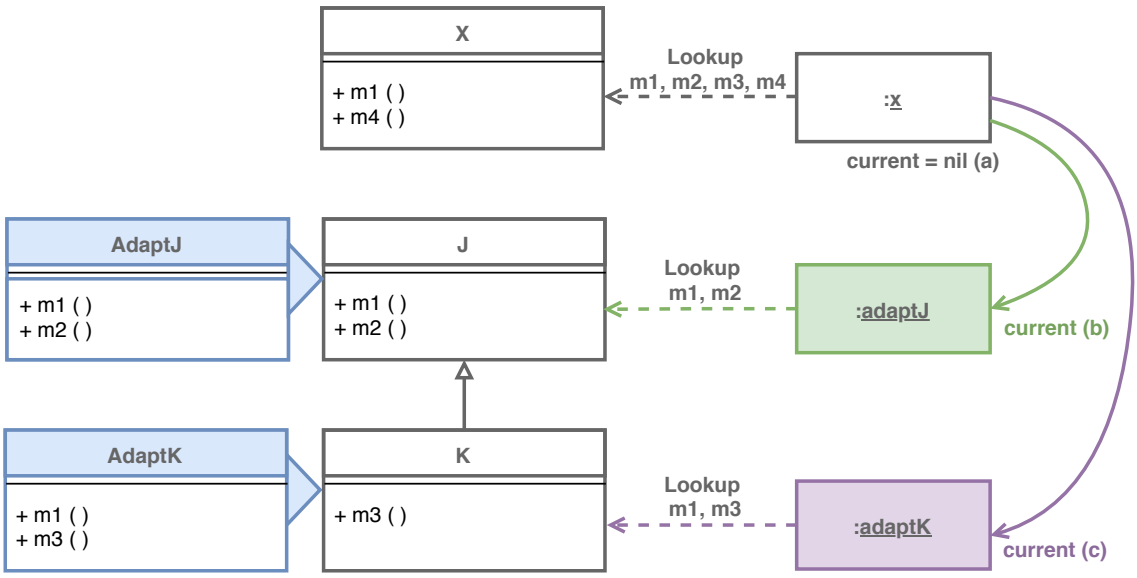


FIGURE 3.8 – **(a)** x n'est pas adapté, son lookup démarre dans son méta-objet (sa classe). **(b)** x est adapté par une instance de *LUBRecord adaptJ*, instance de l'adaptation *AdaptJ*. Le lookup pour les méthodes sélectionnées (*m1, m2*) démarre alors dans la classe cible de l'adaptation (J). Le lookup pour les méthodes non adaptées démarre dans le méta-objet d'origine de x. **(c)** Une nouvelle adaptation *adaptK* est associée à x pour les méthodes *m1, m3*. Le lookup pour *m1, m3* démarre désormais dans K tandis que pour les méthodes non adaptées (*m2, m4*), le lookup démarre dans X.

	(a)	**(b)**	**(c)**
m1	X.m1	J.m1	J.m1
m2	ERREUR	J.m2	ERREUR
m3	ERREUR	ERREUR	K.m3
m4	X.m4	X.m4	X.m4

TABLE 3.1 – Résultats du lookup suivant les scénarios *(a)*, *(b)* et *(c)* de la Figure 3.8

Scénario (a) Il s'agit du cas nominal d'un langage orienté objet : le lookup est démarré dans la classe de l'objet (ici X). Les lookups pour les méthodes *X.m1* et *X.m4* ont été résolus dans la classe X, puisque ces dernières font partie des opérations de X. En revanche, le lookup échoue pour *m2* et *m3* puisque ces opérations ne font pas partie de X.

Scénario (b) L'objet x a été associé à une instance de *Adaptation*, *adaptJ*, qui spécifie un nouveau point de départ du lookup pour les méthodes *m1* et *m2*. Le lookup pour la méthode *m1* est résolu dans la classe J (*J.m1*), et c'est donc ce comportement qui sera exécuté au lieu de celui présent dans la classe X. La méthode *m2* ne fait pas partie des opérations définies dans la classe X. C'est donc un comportement acquis par l'adaptation, et son lookup est également résolu dans la classe J (*J.m2*). Les méthodes *m3* et *m4* ne sont pas concernées par l'adaptation, le lookup est donc résolu dans la classe d'origine de l'objet x (respectivement un échec du lookup pour *m3*, et *X.m4*).

Scénario (c) Une nouvelle adaptation, *adaptK*, est associée à x et spécifie un nouveau point de départ du lookup pour les méthodes *m1, m3*. Le lookup pour *m1* est résolu dans J (*J.m1*), car K est une sous-classe de J et *m1* est définie dans cette dernière. La méthode *m2* ne fait plus partie des opérations acquises, car elle n'est plus spécifiée dans l'adaptation courante *adaptK*. La méthode *m2* ne fait donc plus partie du comportement de l'objet, et le lookup termine par un échec. Le lookup pour *m3* est résolu dans K (*K.m3*) et pour *m4* dans X (*X.m4*).

Observations

Pour les objets non adaptés, représentés dans la structure manipulée par l'environnement d'exécution (c'est-à-dire le modèle d'exécution) par des instances de *ObjectRecord* sans *LUBRecord* dit *courant* (Figure 3.6), le lookup se comporte comme le lookup de base dans les langages comme Smalltalk. C'est ce que l'on observe dans le scénario *(a)*.

C'est également le cas pour les opérations non concernées par l'adaptation d'un objet. Par exemple, dans les scénarios *(b)* et *(c)*, la méthode *m4* n'est pas concernée par les adaptations *adaptJ* et *adaptK*. Le lookup pour *m4* n'est pas impacté, et se comporte comme le lookup du langage hôte (ici *Kernel-Lub*).

Par contraste, les adaptations peuvent cibler des opération spécifiques pour lesquelles le point de départ du lookup est modifié : c'est le cas des méthodes *m1* et *m2* pour l'adaptation *adaptJ* et des méthodes *m1* et *m3* pour *adaptK*.

La méthode *m2* est un cas spécifique, car elle ne fait pas partie du comportement d'origine de l'objet x. Le lookup produit, par son fonctionnement nominal, un échec dans les scénarios *(a)* et *(c)*.

En revanche, dans le scénario *(b)*, un nouveau point de départ du lookup est spécifié pour *m2*. Cette opération devient donc partie du comportement de x tant que l'adaptation qui spécifie le nouveau point de départ du lookup reste active (c'est-à-dire *courante*).

3.3 Collecteurs : groupes d'objets dynamiques pour l'adaptation collective

Lub permet d'adapter un programme avec une granularité objet. Pour appliquer une adaptation à des objets particuliers, il est nécessaire de spécifier quels sont ces objets. Pour que l'adaptation puisse être non-anticipée, il est nécessaire de pouvoir spécifier ces objets pendant l'exécution du programme.

Dans cette section, nous présentons les Collecteurs, qui permettent de spécifier des ensembles d'objets qui seront *collectés* pendant l'exécution du programme. Ces ensembles d'objets peuvent varier dynamiquement, en fonction du contexte d'exécution et de la spécification d'un collecteur. L'objectif est de pouvoir appliquer des opérations ou des transformations aux objets dynamiquement agrégés par un collecteur. Dans le cadre de Lub, l'opération qui nous intéresse est l'opération d'adaptation. Ainsi, il sera possible de spécifier des ensembles d'objets sur lesquels appliquer des adaptations, éventuellement suivant une condition spécifique (une *garde*).

Les Collecteurs sont définis sous la forme d'une adaptation légère de *Kernel-Lub*. Les Collecteurs constituent alors un *patron* d'infrastructure et une sémantique opérationnelle à part, qui peuvent ensuite être combinés avec Lub. Pour en simplifier la lecture, cette section définit directement les collecteurs comme une extension de Lub. Les différences notables seront spécifiées lorsque nécessaire, et en particulier les parties constituant des *patrons* complètement indépendants.

3.3.1 Infrastructure des Collecteurs

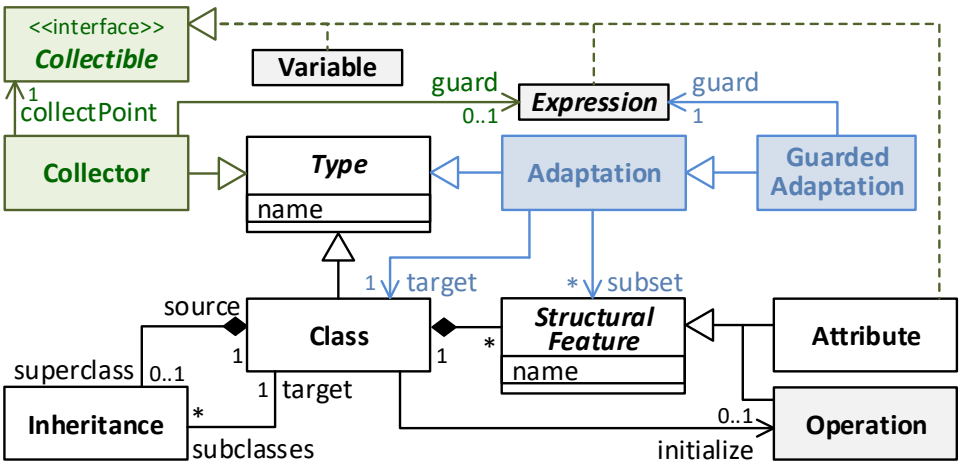


FIGURE 3.9 – Infrastructure de Lub étendue aux Collecteurs

La Figure 3.9 représente le *patron* d'infrastructure de langage de la Figure 3.7 étendu au concept de *Collecteur*.

Un collecteur est un nouveau type du langage (*Collector*). Comme pour une classe ou une adaptation, il s'agit d'un élément nommé. Un collecteur permet de spécifier un ensemble d'objets qui sera mis à jour dynamiquement lors de l'exécution du programme. Dans Lub, ces objets collectés sont destinés à être globalement soumis à une même adaptation. Sur demande, les objets collectés sont soumis à l'adaptation, ou libérés de l'adaptation. Un collecteur peut également être associé à une expression qui joue le rôle de *garde*.

Éléments à collecter. Un collecteur est caractérisé par un point de collecte (*collectPoint*) qui correspond à un élément à collecter (*Collectible*). Un élément à collecter peut être une variable, c'est-à-dire un paramètre formel de méthode ou une variable locale de bloc (Figure 3.2), un attribut ou une expression. Dans les deux premiers cas, les objets collectés correspondent aux objets référencés à l'exécution dans les variables ou attributs concernés. Dans le dernier cas, les objets collectés correspondent aux références valides obtenues lors de l'évaluation de l'expression concernée à l'exécution.

3.3.2 Opérations du langage

Les opérations des Collecteurs, dans le cadre de cette thèse, sont divisées en deux parties. D'abord, les opérations propres au concept de Collecteur, qui étendent *Kernel-Lub*. Ensuite, les opérations de Lub étendues au concept de Collecteurs.

La Figure 3.10 représente le *patron* d'opérations et d'expressions de la Figure 3.5 étendu au concept de *Collecteur*.

3.3.2.1 Opérations des Collecteurs

Le concept de Collecteur n'a pas en lui-même d'impact sur le modèle d'opérations de Lub ou de *Kernel-Lub*. L'extension de *Kernel-Lub* au concept de Collecteur (Figure 3.10) définit en revanche un nouveau type d'envoi de message : *CollectorInstantiation*. Le receveur est un méta-objet qui désigne un collecteur. Les paramètres de *CollectorInstantiation* sont une expression qui joue le rôle de *point de collecte* (obligatoire), ainsi que, de manière optionnelle, une expression qui joue le rôle de *garde* et une stratégie de gestion en mémoire des objets collectés.

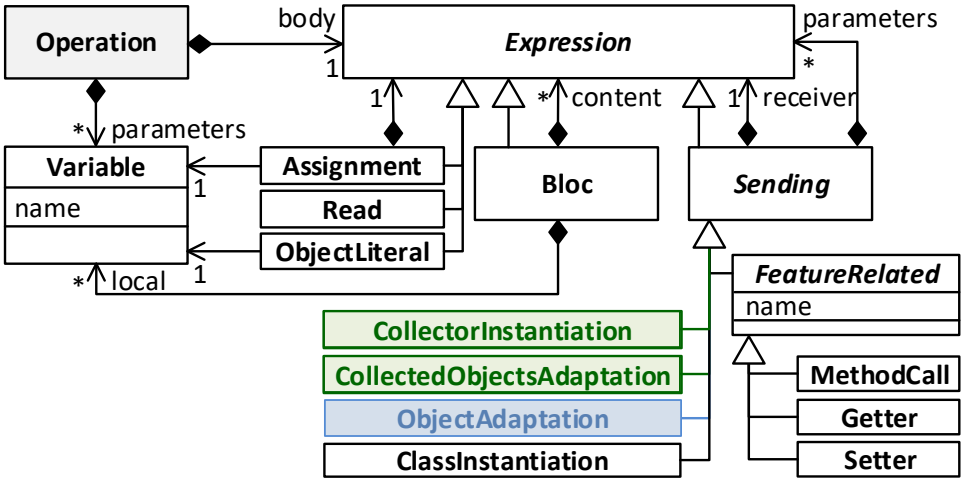


FIGURE 3.10 – Opérations de Lub étendues aux Collecteurs

3.3.2.2 Opérations de Lub étendues aux Collecteurs

Un nouveau type d'envoi de message est rajouté au modèle initial : *CollectAdaptation* (Figure 3.10). Le receveur est un méta-objet qui désigne un collecteur. Le premier paramètre de *CollectAdaptation* (obligatoire) est un méta-objet qui désigne l'adaptation à appliquer à tous les objets collectés par le receveur.

3.3.3 Modèle d'exécution

L'ajout du concept de collecteur n'a pas d'incidence sur le modèle d'exécution décrit par la Figure 3.6.

3.3.4 Sémantique opérationnelle

La sémantique opérationnelle des Collecteurs, dans le cadre de cette thèse, est divisée en deux parties. D'abord, la sémantique opérationnelle propre au concept de Collecteur. Ensuite, la sémantique opérationnelle de Lub étendue au concept de Collecteurs.

3.3.4.1 Sémantique opérationnelle des Collecteurs

Création d'un collecteur. Lors de l'initialisation d'un collecteur (*CollectorInstantiation*), un ensemble initialement vide est créé. Lorsqu'une expression ciblée par un *collectPoint* (Figure 3.9) est évaluée à l'exécution, le

résultat de cette évaluation peut être collecté :

— Si le collecteur est soumis à une garde, elle est alors évaluée. Si le résultat ne correspond pas à la référence du littéral *true*, la collecte est inactive et l'ensemble de références n'est pas modifié.

— Si le collecteur n'est pas soumis à une garde, ou s'il est soumis à une garde dont l'évaluation correspond à la référence du littéral *true*, un objet peut être collecté. Il correspond selon le type de *collectPoint* à la référence d'une instance contenue dans une variable ou un attribut, ou à la référence d'une instance obtenue par évaluation d'une expression. L'objet collecté est ajouté à l'ensemble de références du collecteur.

Mise à jour des objets collectés. L'ensemble initial des objets collectés à la création du collecteur évolue en fonction de la mise à jour du point de collecte. Selon les cas, cette mise à jour est provoquée par la mise à jour de la référence stockée dans l'attribut ou la variable concernée, ou par la mise à jour d'un des composants de l'expression concernée. En cas de mise à jour du point de collecte, l'éventuelle garde est à nouveau évaluée et la nouvelle référence est ajoutée à l'ensemble des objets collectés.

Gestion mémoire des objets collectés. Il n'y a pas de contrainte sur la gestion mémoire des ensembles d'objets collectés par les collecteurs. Trois stratégies sont définies, et sont mutuellement exclusives. Ces stratégies peuvent être dynamiquement interchangées pendant l'exécution, indépendamment du moment où les objets entrent ou sortent d'un collecteur.

— **Conservation des objets.** Les objets collectés sont conservés en mémoire [1] indépendamment de la stratégie de gestion de la mémoire du programme en cours d'exécution. Les objets ne peuvent pas être réclamés par un gestionnaire de mémoire tant qu'ils sont référencés par un collecteur.
— **Conservation par référence faible.** La gestion des objets collectés est laissée au contrôle du gestionnaire de mémoire sans interférer avec celui-ci. Lorsqu'un objet est réclamé par le gestionnaire de la mémoire du programme en cours d'exécution pour être libéré, il sort de l'ensemble des objets collectés.
— **Conservation par copie.** Les objets collectés sont copiés, et ces copies sont stockées en mémoire [1]. La gestion des objets originaux est laissée au contrôle du gestionnaire de mémoire sans interférer avec celui-ci. Les copies de ces objets ne peuvent pas être réclamés par un gestionnaire de mémoire tant qu'ils sont référencés par un collecteur.

1. "*en mémoire*" signifie ici "dans la mémoire du programme"

La stratégie active par défaut est la conservation des objets par copie. En cas de mise à jour dynamique de la stratégie de gestion mémoire du collecteur, la nouvelle stratégie est immédiatement appliquée aux objets déjà collectés.

3.3.4.2 Lub : Sémantique opérationnelle étendue aux Collecteurs

La sémantique opérationnelle de Lub étendue au concept de collecteur est obtenue par traduction vers la sémantique opérationnelle de Lub étendue au concept d'adaptation (cf. section 3.2.4). Concrètement, un collecteur permet de rassembler des objets auxquels une adaptation peut être globalement appliquée comme si elle était individuellement appliquée à chacun des objets rassemblés.

Adaptation des objets collectés. À l'exécution de *CollectAdaptation*, un ensemble d'objets d'un collecteur particulier est immédiatement adapté grâce à l'adaptation a fournie en paramètre, comme si chaque objet avait été adapté individuellement par l'appel de *ObjectAdaptation* avec en paramètre l'adaptation a.

Mise à jour des objets collectés. Si une adaptation a été appliquée sur un ensemble d'objets collectés via *CollectAdaptation*, l'adaptation de chaque objet entrant ou sortant du groupe d'objets est automatiquement mise à jour.

Gestion mémoire des objets collectés. La stratégie de gestion mémoire appliquée par l'extension de Lub au concept de collecteur est la stratégie de conservation des objets par références faibles. La collection des objets n'est utilisée que pour obtenir des références à ces derniers et leur appliquer une adaptation de comportement. Un appel à *CollectAdaptation* sur un collecteur le bascule immédiatement sur cette stratégie, si cette dernière n'était pas déjà active.

Mise à jour de l'adaptation. L'adaptation d'un collecteur existant peut être mise à jour par un nouvel appel à *CollectAdaptation* avec en paramètre un méta-objet qui désigne une nouvelle adaptation à appliquer aux objets, ainsi qu'une éventuelle garde. Lorsque ce paramètre correspond à *null*, les objets collectés ne sont plus adaptés comme s'ils avaient été réinitialisés individuellement par l'appel de *ObjectAdaptation* avec en paramètre le littéral *null*.

Lorsque des objets rentrent ou sortent d'un ensemble défini par un collecteur, leur adaptation est automatiquement mise à jour :

— Un objet sortant du groupe est soit adapté avec le littéral *null* et retrouve son comportement originel, ou renversé vers une adaptation précédente et

postérieure à son entrée dans le groupe d'objets ou à sa première adaptation dans le groupe d'objets par *CollectAdaptation*.

— Un objet entrant dans le groupe est automatiquement adapté avec la dernière adaptation a fournie précédemment en paramètre du dernier appel à *CollectAdaptation* (avec sa garde éventuelle et suivant le résultat de son évaluation).

3.3.5 La collecte dynamique d'objets : une illustration

Cette section illustre la collection d'objets à partir d'un point de collecte, défini sur une expression d'un programme. Ce programme est constitué d'une seule instruction, qui s'exécute en boucle, et dont le code en Smalltalk est présenté par la Figure 3.11. Dans cet exemple, le code présenté s'exécute une fois toutes les minutes. L'instruction de la ligne 1 instancie un objet de la classe $Time$ pour obtenir l'heure courante, et l'imprime via l'instruction *printString*. Nous supposons ici que *printString* écrit dans une console l'heure contenue dans l'instance de $Time$, dans un format lisible par l'utilisateur-rice.

Initialisation du collecteur. Nous souhaitons collecter les instances de $Time$, chaque fois que l'expression *Time now* est exécutée. Un collecteur peut être défini comme suit, avec l'instruction $CollectorInstantiation$:

— Point de collecte (*collectPoint*) : l'expression "*Time now*".
— Garde : $\emptyset$.
— Ensemble d'objets collectés : { } (aucun objet).
— Stratégie de gestion mémoire : conservation par copie.

Cette définition de collecteur est effectuée alors même que le programme est en cours d'exécution, sans l'interrompre. La non interruption de programme est considérée ici comme étant la possibilité de créer un collecteur sans avoir à redémarrer le programme. L'exécution du programme peut éventuellement être mise en pause − ou ralentie − par les opérations d'initialisation et d'installation du collecteur.

```
1 Time now printString
```

FIGURE 3.11 – Programme illustratif pour la collecte d'objets. Nous souhaitons collecter les objets instances de $Time$ chaque fois qu'ils sont crées.

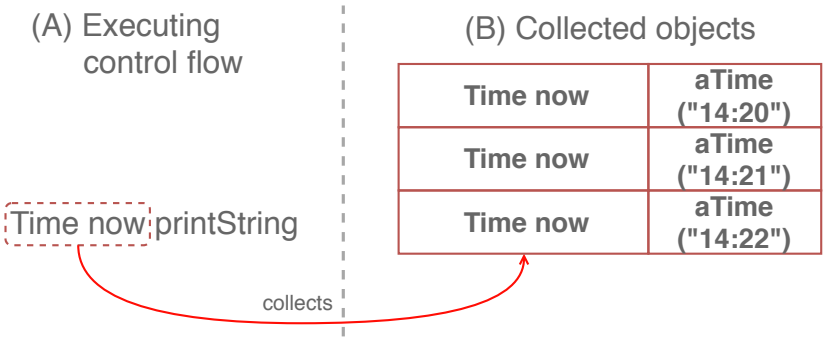


FIGURE 3.12 – Illustration d'un collecteur : le *collectPoint* est une expression particulière du programme *Time now* (A), dont le résultat de l'évaluation sera collecté et stocké par le collecteur (B).

Résultat de l'exécution du programme. Une fois le collecteur initialisé, celui-ci collecte les objets issus de l'évaluation du point de collecte – c'est-à-dire l'expression *Time now*. La Figure 3.12 montre le point de collecte (A) et le résultat de la collecte (B). Chaque fois que l'exécution du programme "passe" dans le flot de contrôle du point de collecte, c'est-à-dire l'expression *Time now*, le résultat de son évaluation est collecté. Conformément à la stratégie de la gestion mémoire sélectionnée, les instances de *Time* sont copiées et stockées par le collecteur dans la mémoire du programme en cours d'exécution. Ce stockage s'effectue de manière ordonnée, suivant l'ordre de collecte qui dépend du flot de contrôle.

La collecte s'effectue à partir de l'évaluation du point de collecte. Dans le flot de contrôle, la collecte est donc réalisée immédiatement après l'évaluation de l'expression *Time now*, mais avant que le message *printString* soit envoyé à l'objet résultant de cette évaluation. L'instance de *Time* est un objet éphémère et le gestionnaire de mémoire le réclame une fois qu'il n'est plus référencé, c'est-à-dire après l'exécution de la méthode *printString*. Les copies des objets collectés persistent cependant dans le collecteur, conformément à la stratégie de conservation des objets par copie. Le résultat de l'évaluation de l'expression complète produit la chaîne de caractères suivante dans la console : *14:22*.

Application d'adaptation de comportement aux objets collectés. Nous souhaitons désormais appliquer une adaptation de comportement aux instances de *Time* qui sont créées dans le flot de contrôle de la Figure 3.11. L'objectif est d'adapter la méthode *printString* pour changer la visualisation de l'heure dans la console. Nous concevons une adaptation de comportement qui remplace le caractère ":" par le caractère "*h*" dans la méthode *printString*. L'opération

CollectedObjectsAdaptation, avec cette adaptation en paramètre, est appliquée au collecteur et a les effets suivants :

— À chaque collecte, l'instance de *Time* nouvellement créée entre dans le groupe d'objets du collecteur. Comme une adaptation a été appliquée au collecteur, ce dernier applique immédiatement l'adaptation à l'objet collecté. La suite de l'exécution est donc l'exécution de la méthode adaptée *printString*, et affiche dans la console la chaîne de caractères "14*h*23" (au lieu de "*14*:*23*"). Les autres instances de la classe *Time* en dehors du flot de contrôle du point de collecte ne seront pas modifiés. Seuls les objets collectés se voient appliquer l'adaptation de comportement.

— La stratégie de gestion mémoire de conservation par références faibles est forcée par l'application de l'opération *CollectedObjectsAdaptation*. Les objets ne sont plus ni copiés, ni stockés, mais relâchés dès que le gestionnaire de mémoire les réclame.

3.4 Propriétés fondamentales

Dans cette section, une sémantique dénotationnelle unifiée de l'infrastructure du langage et de l'environnement d'exécution est définie dans le but de spécifier le *lookup* et son extension aux principes d'adaptation. Cette sémantique dénotationnelle est fondée sur la notion de graphe. La sémantique opérationnelle qui décrit l'évolution de l'environnement d'exécution en fonction des éléments d'un programme en cours d'exécution sort du cadre de ces travaux dont l'objectif principal est l'étude de l'adaptabilité d'un programme grâce à une altération légère des principes du *lookup*. Cette sémantique opérationnelle n'est donc pas formalisée ici.

3.4.1 Kernel-Lub : formalisation du lookup

3.4.1.1 Graphe de programme

Un graphe de programme est une *dénotation de spécification de programme* basée sur la notion de graphe. Il s'agit plus précisément d'un graphe orienté et non-étiqueté composé de sommets qui représentent des classes, et d'arcs qui représentent des méthodes et des liens d'héritage. La Figure 3.13 introduit les *espaces de noms* utiles pour la définition d'un graphe de programme et de son extension aux systèmes d'objets définis dans la section suivante. Ces espaces de noms sont des *alphabets*, *i.e.* des ensembles finis non-vides de symboles.

On appelle *programme* et on note $\mathfrak{p}$ le triplet composé d'un ensemble fini de classes noté $\mathfrak{p}_{\mathcal{C}}$ correspondant aux sommets d'un graphe, et de deux ensembles d'arcs notés $\mathfrak{p}_{\mathcal{M}}$ et $\mathfrak{p}_{\Delta}$ (cf. Figure 3.14). Le premier ensemble d'arcs dénote les méthodes. Le

deuxième est caractérisé par une fonction totale sur $\mathfrak{p}_\mathcal{C}$ qui dénote les liens d'héritage *simple*.

$$\mathcal{C} : \text{classes} \qquad \mathcal{M} : \text{méthodes} \qquad \mathcal{I} : \text{instances}$$

FIGURE 3.13 – Espaces de noms

$$\mathfrak{p} \triangleq \left(\mathfrak{p}_\mathcal{C} \ , \ \mathfrak{p}_\mathcal{M} \ , \ \mathfrak{p}_\Delta \right)$$

$$\text{avec} \left\{ \begin{array}{rcl} \mathfrak{p}_\mathcal{C} & \subseteq & \mathcal{C} \\ \mathfrak{p}_\mathcal{M} & \subseteq & \mathfrak{p}_\mathcal{C} \times \mathcal{M} \\ \mathfrak{p}_\Delta & : & \mathfrak{p}_\mathcal{C} \to \mathfrak{p}_\mathcal{C} \cup \{\bot\} \end{array} \right.$$

FIGURE 3.14 – Définition de *graphe de programme*

La fonction $\mathfrak{p}_\Delta$ permet de définir pour chaque classe l'éventuelle classe dont elle hérite ($\bot$ si elle n'hérite pas de classe). La fonction $\mathfrak{p}^*_\Delta$ définie dans la Figure 3.15 fournit pour chaque classe d'un programme $\mathfrak{p}$ l'éventuelle classe correspondant au $n^{\text{ème}}$ ancêtre dans sa hiérarchie d'héritage ($\bot$ si elle n'existe pas). Cette fonction permet de définir la propriété de validité de l'héritage qui interdit les cycles.

$$\mathfrak{p}^*_\Delta : \left| \begin{array}{rcl} (\mathfrak{p}_\mathcal{C} \times \mathbb{N}^*) & \to & \mathfrak{p}_\mathcal{C} \cup \{\bot\} \\ (c, 1) & \mapsto & \mathfrak{p}_\Delta(c) \\ (c, n) & \mapsto & \left| \begin{array}{l} \bot \text{ si } \mathfrak{p}_\Delta(c) = \bot \\ \mathfrak{p}^*_\Delta(\mathfrak{p}_\Delta(c), n-1) \text{ sinon} \end{array} \right. \end{array} \right.$$

$$\text{validité} : \ \forall c \in \mathfrak{p}_\mathcal{C}, \exists n \in \mathbb{N}^*, \ \mathfrak{p}^*_\Delta(c, n) = \bot$$

FIGURE 3.15 – Propriété de l'héritage simple

Un graphe de programme est une abstraction de lot d'instances conforme au modèle d'infrastructure de la Figure 3.1. Cette abstraction destinée à la formalisation du *lookup* ne tient pas compte des attributs et des fonctions d'initialisation.

3.4.1.2 Graphe de système d'objets

Un graphe de système d'objets est une *dénotation de programme en cours d'exécution* basée sur la notion de graphe de programme étendu aux instances. Dans cette extension, de nouveaux sommets représentent des instances, et de nouveaux arcs représentent des liens d'introspection.

On appelle *système d'objets* et on note $\mathfrak{s}$ le triplet composé d'un graphe de programme noté $\mathfrak{s}_\mathfrak{p}$, d'un ensemble fini d'instances noté $\mathfrak{s}_\mathcal{I}$, et d'une fonction totale de $\mathfrak{s}_\mathcal{I}$ notée $\mathfrak{s}_{\text{ meta}}$ (cf. Figure 3.16).

$$\mathfrak{s} \triangleq \left(\mathfrak{s}_\mathfrak{p} \ , \ \mathfrak{s}_\mathcal{I} \ , \ \mathfrak{s}_{\text{ meta}} \right) \text{ avec } \begin{cases} \mathfrak{s}_\mathcal{I} & \subseteq & \mathcal{I} \\ \mathfrak{s}_{\text{ meta}} & : & \mathfrak{s}_\mathcal{I} \rightarrow \mathfrak{s}_{\mathfrak{p}_\mathcal{C}} \end{cases}$$

FIGURE 3.16 – Définition de *graphe de système d'objets*

Le graphe de programme dénote la spécification du programme en cours d'exécution. L'ensemble d'instances dénote les objets physiquement présents en mémoire au moment d'exécution que le système d'objets représente. La fonction $\mathfrak{s}_{\text{ meta}}$ dénote les liens d'introspection qui relient chaque instance à exactement une classe du programme. Plus précisément, elle permet de définir pour chaque objet du système l'unique classe du programme dont elle est une instance.

Un graphe de système d'objets est une abstraction de lot d'instances conforme au modèle d'environnement d'exécution de la Figure 3.3. Cette abstraction destinée à la formalisation du *lookup* ne tient pas compte des slots, des valeurs primitives et des références. Le rôle metaObject est dénoté par la fonction $\mathfrak{s}_{\text{ meta}}$ qui fait le lien avec la spécification d'un programme en cours d'exécution.

3.4.1.3 Spécification du *lookup*

Le *lookup* est l'opération qui consiste à déterminer la classe contenant la définition d'une méthode qui doit être exécutée au moment de son envoi à un receveur donné.

Dans le cadre formel des graphes, le *lookup* est défini par les fonctions $\text{lookup}_\mathfrak{p}$ et $\text{lookup}_\mathfrak{s}$, respectivement dépendantes d'un programme $\mathfrak{p}$ et d'un système d'objets $\mathfrak{s}$ (cf. Figure 3.17).

La fonction $\text{lookup}_\mathfrak{p}$ détermine un chemin dans le graphe d'héritage de $\mathfrak{p}$. La fonction $\text{lookup}_\mathfrak{s}$ détermine un point de départ du *lookup* en fonction d'une instance de $\mathfrak{s}$ et d'un paramètre booléen $b \in \mathbb{B} \triangleq \{t, f\}$, qui indique si le point de départ doit être une super-classe ou non.

$$\begin{array}{rcrcl} \text{lookup}_{\mathfrak{p}} & : & (\mathfrak{p}_{\mathcal{C}} \times \mathcal{M}) & \rightarrow & \mathfrak{p}_{\mathcal{C}} \cup \{\bot\} \\ \text{lookup}_{\mathfrak{s}} & : & (\mathfrak{s}_{\mathcal{I}} \times \mathcal{M} \times \mathbb{B}) & \rightarrow & \mathfrak{s}_{\mathfrak{p}_{\mathcal{C}}} \cup \{\bot\} \end{array}$$

FIGURE 3.17 – Fonctions de *lookup*

Les propriétés caractéristiques de $\text{lookup}_{\mathfrak{p}}$ et de $\text{lookup}_{\mathfrak{s}}$ sont décrites dans la Figure 3.18. Les propriétés ①, ② et ③ concernent le point de départ du *lookup* (propriétés de $\text{lookup}_{\mathfrak{s}}$). Les propriétés ④, ⑤ et ⑥ concernent le parcours des liens d'héritage (propriétés de $\text{lookup}_{\mathfrak{p}}$).

$$\begin{array}{l} \forall (i,m) \in \mathfrak{s}_{\mathcal{I}} \times \mathcal{M}, \\ \quad ① \quad \text{lookup}_{\mathfrak{s}}(i,m,f) = \text{lookup}_{\mathfrak{s}_{\mathfrak{p}}}(\mathfrak{s}_{\text{ meta}}(i),m) \\ \quad ② \quad \exists c \in \mathfrak{s}_{\mathfrak{p}_{\mathcal{C}}}, \mathfrak{s}_{\mathfrak{p}_{\Delta}}(\mathfrak{s}_{\text{ meta}}(i)) = c \implies \\ \qquad \text{lookup}_{\mathfrak{s}}(i,m,t) = \text{lookup}_{\mathfrak{s}_{\mathfrak{p}}}(c,m) \\ \quad ③ \quad \mathfrak{s}_{\mathfrak{p}_{\Delta}}(\mathfrak{s}_{\text{ meta}}(i)) = \bot \implies \text{lookup}_{\mathfrak{s}}(i,m,t) = \bot \end{array}$$

$$\begin{array}{l} \forall (c,m) \in \mathfrak{p}_{\mathcal{C}} \times \mathcal{M}, \\ \quad ④ \quad (c,m) \in \mathfrak{p}_{\mathcal{M}} \implies \text{lookup}_{\mathfrak{p}}(c,m) = c \\ \quad ⑤ \quad (c,m) \notin \mathfrak{p}_{\mathcal{M}} \wedge \exists c' \in \mathfrak{p}_{\mathcal{C}}, \mathfrak{p}_{\Delta}(c) = c' \implies \\ \qquad \text{lookup}_{\mathfrak{p}}(c,m) = \text{lookup}_{\mathfrak{p}}(c',m) \\ \quad ⑥ \quad (c,m) \notin \mathfrak{p}_{\mathcal{M}} \wedge \mathfrak{p}_{\Delta}(c) = \bot \implies \\ \qquad \text{lookup}_{\mathfrak{p}}(c,m) = \bot \end{array}$$

FIGURE 3.18 – Propriétés du *lookup*

Dans le cas général, le point de départ du *lookup* correspond à la classe dont l'objet i à qui le message m est envoyé est une instance (propriété ①). Dans le cas particulier où le receveur de m est spécifié dans le programme par *super*, le point de départ du *lookup* correspond à la super-classe de l'objet i à qui le message m est envoyé (propriété ②), à condition que cette super-classe existe (propriété ③).

Le parcours des liens d'héritage d'une classe s'arrête dès que la méthode recherchée apparaît dans la classe (propriété ④). Si ce n'est pas le cas, le parcours continue dans sa super-classe (propriété ⑤), à condition que cette super-classe existe (propriété ⑥).

3.4.2 Lub : formalisation du lookup

Le cadre formel des graphes de programmes et de systèmes d'objet introduit dans la section 3.4.1 est étendu à la notion d'adaptation.

3.4.2.1 Graphe de programme adaptable

La Figure 3.19 étend les espaces de noms introduits dans la Figure 3.13 à l'espace des noms d'adaptation noté $\mathcal{A}$.

$$\begin{array}{lcllcl} \mathcal{C} & : & \text{classes} & \mathcal{M} & : & \text{méthodes} \\ \mathcal{I} & : & \text{instances} & \mathcal{A} & : & \text{adaptations} \end{array}$$

FIGURE 3.19 – Noms d'adaptations

La Figure 3.20 étend la définition d'un programme de la Figure 3.14 aux adaptations notées $\mathfrak{p}_{\mathcal{A}}$ et aux liens notés $\mathfrak{p}_{\text{ adapt}}$ qui les relient aux classes.

$$\mathfrak{p} \triangleq \left(\mathfrak{p}_{\mathcal{C}} \ , \ \mathfrak{p}_{\mathcal{M}} \ , \ \mathfrak{p}_{\Delta} \ , \ \mathfrak{p}_{\mathcal{A}} \ , \ \mathfrak{p}_{\text{ adapt}} \right)$$

$$\text{avec} \left\{ \begin{array}{rcl} \mathfrak{p}_{\mathcal{A}} & \subseteq & \mathcal{A} \\ \mathfrak{p}_{\text{ adapt}} & : & \mathfrak{p}_{\mathcal{A}} \rightarrow (\mathfrak{p}_{\mathcal{C}} \times \mathcal{P}(\mathcal{M})) \end{array} \right.$$

FIGURE 3.20 – Graphe de programme adaptable

La fonction $\mathfrak{p}_{\text{ adapt}}$ associe à chaque adaptation du programme un couple comportant une classe du programme et un ensemble éventuellement vide de méthodes. La Figure 3.21 introduit la contrainte de validité des adaptations qui garantit que les méthodes ciblées par une adaptation sont bien des méthodes de la classe adaptée.

Un graphe de programme adaptable est une abstraction de lot d'instances conforme au modèle d'infrastructure de la Figure 3.4.

3.4.2.2 Système d'objets adaptable

La Figure 3.22 étend la définition d'un système d'objets de la Figure 3.16 aux liens notés $\mathfrak{s}_{\text{ adapt}}$ qui relient les instances du système $\mathfrak{s}$ aux adaptations du programme associé $\mathfrak{s}_{\mathfrak{p}}$.

$$\forall (a, c, E) \in \mathfrak{p}_{\mathcal{A}} \times \mathfrak{p}_{\mathcal{C}} \times \mathcal{P}(\mathcal{M}),$$
$$\mathfrak{p}_{\text{ adapt}}(a) = (c, E) \implies \forall m \in E, (c, m) \in \mathfrak{p}_{\mathcal{M}}$$

FIGURE 3.21 – Validité des adaptations

$$\mathfrak{s} \triangleq \left(\mathfrak{s}_{\mathfrak{p}} \ , \ \mathfrak{s}_{\mathcal{I}} \ , \ \mathfrak{s}_{\text{ meta}} \ , \ \mathfrak{s}_{\text{ adapt}} \right)$$
$$\text{avec } \mathfrak{s}_{\text{ adapt}} : \mathfrak{s}_{\mathcal{I}} \to (\mathfrak{s}_{\mathfrak{p}_{\mathcal{A}}} \cup \bot)$$

FIGURE 3.22 – Système d'objets adaptable

La fonction $\mathfrak{s}_{\text{ adapt}}$ associe à chaque instance du système une adaptation du programme, auquel cas l'instance est adaptée, ou bien $\bot$ dans le cas où l'instance n'est pas adaptée.

Un graphe de système d'objets est une abstraction de lot d'instances conforme au modèle d'environnement d'exécution de la Figure 3.6.

3.4.2.3 Spécification du *lookup* adaptable

$$\text{lookup}_{\mathfrak{s}_{\text{ adapt}}} : (\mathfrak{s}_{\mathcal{I}} \times \mathcal{M} \times \mathbb{B}) \to \mathfrak{s}_{\mathfrak{p}_{\mathcal{C}}} \cup \{\bot\}$$

FIGURE 3.23 – Fonction de *lookup* adapté

Les fonctions $\text{lookup}_{\mathfrak{p}}$ et $\text{lookup}_{\mathfrak{s}}$ introduites dans la Figure 3.17 sont complétées par la fonction $\text{lookup}_{\mathfrak{s}_{\text{ adapt}}}$ de la Figure 3.23.

La fonction $\text{lookup}_{\mathfrak{p}}$ détermine un chemin dans le graphe d'héritage de $\mathfrak{p}$ et reste inchangée. La fonction $\text{lookup}_{\mathfrak{s}}$ détermine un point de départ du *lookup* sans prise en compte de l'adaptation. La fonction $\text{lookup}_{\mathfrak{s}_{\text{ adapt}}}$ détermine également un point de départ du *lookup*, mais en tenant compte de l'adaptation.

Les propriétés de la Figure 3.18 concernent les fonctions $\text{lookup}_{\mathfrak{p}}$ et $\text{lookup}_{\mathfrak{s}}$. Elle restent donc inchangées. Elles sont complétées par les propriétés de la Figure 3.24 qui concernent la fonction $\text{lookup}_{\mathfrak{s}_{\text{ adapt}}}$.

Dans le cas d'une instance i adaptée, $\mathfrak{s}_{\text{ adapt}}(i)$ fournit son adaptation a, et $\mathfrak{s}_{\mathfrak{p}_{\text{ adapt}}}(a)$ donne les détails de cette adaptation. Si la méthode m recherchée fait

partie de cette adaptation et que le *lookup* de cette méthode à partir de la classe c ciblée par cette adaptation fournit une classe c', alors le résultat du *lookup* pour i est dans ce cas la classe c' (propriété ⑦). Cette propriété détermine toutes les conditions à réunir pour que l'adaptation soit effective.

Dans le cas d'une instance i adaptée, dont l'adaptation ne concerne pas la méthode m recherchée, ou dont le *lookup* à partir de la classe c ciblée par l'adaptation n'aboutit pas, le *lookup* non adapté s'applique (propriété ⑧).

Enfin, dans le cas d'une instance i non adaptée, le *lookup* non adapté s'applique (propriété ⑨).

$$
\begin{array}{l}
\forall (i, m, b) \in \mathfrak{s}_{\mathcal{I}} \times \mathcal{M} \times \mathbb{B}, \\
\exists (a, c, E) \in \mathfrak{s}_{\mathfrak{p}_{\mathcal{A}}} \times \mathfrak{s}_{\mathfrak{p}_{\mathcal{C}}} \times \mathcal{P}(\mathcal{M}), \\
(\mathfrak{s}_{\text{ adapt}}(i) = a \wedge \mathfrak{s}_{\mathfrak{p}\text{ adapt}}(a) = (c, E)) \implies \\
\quad ⑦ \quad m \in E \wedge \exists c' \in \mathfrak{s}_{\mathfrak{p}_{\mathcal{C}}},\ \text{lookup}_{\mathfrak{s}_{\mathfrak{p}}}(c, m) = c' \implies \\
\qquad \text{lookup}_{\mathfrak{s}\text{ adapt}}(i, m, b) = c' \\
\quad ⑧ \quad m \notin E \vee\ \text{lookup}_{\mathfrak{s}_{\mathfrak{p}}}(c, m) = \bot \implies \\
\qquad \text{lookup}_{\mathfrak{s}\text{ adapt}}(i, m, b) = \text{lookup}_{\mathfrak{s}}(i, m, b)
\end{array}
$$

$$
\begin{array}{l}
\forall (i, m, b) \in \mathfrak{s}_{\mathcal{I}} \times \mathcal{M} \times \mathbb{B}, \\
\quad ⑨ \quad \mathfrak{s}_{\text{ adapt}}(i) = \bot \implies \\
\qquad \text{lookup}_{\mathfrak{s}\text{ adapt}}(i, m, b) = \text{lookup}_{\mathfrak{s}}(i, m, b)
\end{array}
$$

FIGURE 3.24 – Propriétés du *lookup* adaptable

3.4.2.4 Propriété fondamentale

Le *lookup* adapté permet à un objet de réagir ponctuellement différemment à la réception d'un message. Il peut en particulier réagir à un message auquel il ne pouvait pas répondre avant l'adaptation. Il s'agit dans ce cas d'une *extension* de l'interface de l'objet.

$$
\begin{array}{l}
\forall (i, m, b) \in \mathfrak{s}_{\mathcal{I}} \times \mathcal{M} \times \mathbb{B}, \\
\text{lookup}_{\mathfrak{s}}(i, m, b) \neq \bot \implies \text{lookup}_{\mathfrak{s}\text{ adapt}}(i, m, b) \neq \bot
\end{array}
$$

FIGURE 3.25 – Non réduction d'interface

En revanche, le *lookup* adapté interdit la *réduction* d'interface. De cette manière, les risques de régression sont limités. Un objet initialement capable de répondre à un message reste dans tous les cas d'adaptation capable d'y répondre.

Le cadre formel des graphes de programmes et de systèmes d'objet permet d'énoncer cette propriété (cf. Figure 3.25) et de la démontrer (cf. Figure 3.26).

La propriété de non réduction d'interface énonce en substance que si le *lookup* natif d'une méthode sur une instance n'échoue pas, alors le *lookup* adapté de cette même méthode sur la même instance ne peut pas échouer non plus.

Preuve *Soit $\mathfrak{s}$, un système d'objets adaptable quelconque. Soient $i \in \mathfrak{s}_{\mathcal{I}}$, une instance quelconque de $\mathfrak{s}$, $m \in \mathcal{M}$, une méthode quelconque, et $b \in \mathbb{B}$, une valeur booléenne quelconque.*

On démontre la proposition contraposée de la propriété de non réduction d'interface :

$$lookup_{\mathfrak{s}\ adapt}(i, m, b) = \bot \implies lookup_{\mathfrak{s}}(i, m, b) = \bot$$

On suppose que $lookup_{\mathfrak{s}\ adapt}(i, m, b) = \bot$. Il faut donc démontrer que dans tous les cas, $lookup_{\mathfrak{s}}(i, m, b) = \bot$.

Cas 1 : i est une instance adaptée. Donc : $\exists (a, c, E) \in \mathfrak{s}_{\mathfrak{p}_{\mathcal{A}}} \times \mathfrak{s}_{\mathfrak{p}_{\mathcal{C}}} \times \mathcal{P}(\mathcal{M}), (\mathfrak{s}_{\ adapt}(i) = a \wedge \mathfrak{s}_{\mathfrak{p}\ adapt}(a) = (c, E))$.
La contraposée de la propriété ⑦ énonce : $lookup_{\mathfrak{s}\ adapt}(i, m, b) \neq lookup_{\mathfrak{s}_{\mathfrak{p}}}(c, m) \implies m \notin E \vee \forall c' \in \mathfrak{s}_{\mathfrak{p}_{\mathcal{C}}}, lookup_{\mathfrak{s}_{\mathfrak{p}}}(c, m) \neq c'$. Plus simplement : $lookup_{\mathfrak{s}\ adapt}(i, m, b) \neq lookup_{\mathfrak{s}_{\mathfrak{p}}}(c, m) \implies m \notin E \vee lookup_{\mathfrak{s}_{\mathfrak{p}}}(c, m) = \bot$.
Comme $lookup_{\mathfrak{s}\ adapt}(i, m, b) = \bot$, on a : $lookup_{\mathfrak{s}_{\mathfrak{p}}}(c, m) \neq \bot \implies m \notin E \vee lookup_{\mathfrak{s}_{\mathfrak{p}}}(c, m) = \bot$. La simplification de cette implication donne : $m \notin E \vee lookup_{\mathfrak{s}_{\mathfrak{p}}}(c, m) = \bot$.
D'après la propriété ⑧, on a : $lookup_{\mathfrak{s}\ adapt}(i, m, b) = lookup_{\mathfrak{s}}(i, m, b)$, soit : $lookup_{\mathfrak{s}}(i, m, b) = \bot$.

Cas 2 : i est une instance non adaptée. Donc : $\mathfrak{s}_{\ adapt}(i) = \bot$. D'après la propriété ⑨, on a : $lookup_{\mathfrak{s}\ adapt}(i, m, b) = lookup_{\mathfrak{s}}(i, m, b)$. Autrement dit, $lookup_{\mathfrak{s}}(i, m, b) = \bot$.

Dans les deux cas possibles, on a bien : $lookup_{\mathfrak{s}}(i, m, b) = \bot$. □

FIGURE 3.26 – Démonstration de la propriété de non réduction d'interface

3.5 Conclusion

Dans ce chapitre, nous avons défini des patrons de langage pour l'adaptation non-anticipée de comportement à granularité objet. Nous avons d'abord défini un langage objet minimal à typage dynamique, *Kernel-Lub* : son infrastructure, ses opérations, son modèle d'exécution et sa sémantique opérationnelle. Ce langage minimal représente une classe de langages pouvant être étendus avec des adaptations non-anticipées de comportement par les patrons de langage proposés.

Nous avons défini et illustré Lub, une extension de *Kernel-Lub* pour adapter des instances dans un programme à objet. Lors de l'adaptation, le méta-objet à partir duquel le *lookup* démarre est modifié pour produire une indirection du *lookup*. Le résultat du lookup est alors une méthode ou un attribut différent, aboutissant à une adaptation concrète du comportement de l'objet. Nous avons formellement démontré la non réductibilité d'interface lors des indirections du *lookup* dues aux adaptations. La modification du point de départ du *lookup* par Lub ne peut pas provoquer d'erreurs pour un appel de méthode, si cet appel de méthode ne provoque pas d'erreurs dans le cas nominal (c'est-à-dire non adapté). L'adaptation peut également être soumise à une condition pour être effective, sous la forme d'une garde évaluée pour autoriser − ou non − l'indirection de *lookup*.

Nous avons ensuite défini et illustré les Collecteurs, qui agrègent dynamiquement des groupes d'objets. Les collecteurs définissent des points de collecte sur des expressions du programme en cours d'exécution. Lorsque ces expressions sont évaluées, les objets ou valeurs issus de ces évaluations sont agrégés dans des ensembles d'objets contenus dans le collecteur. Différentes stratégies, interchangeables dynamiquement, permettent de conserver ces objets indépendamment du gestionnaire de mémoire du programme ou de lui laisser le contrôle total.

Lub est étendu aux Collecteurs, ce qui permet d'appliquer dynamiquement une adaptation globale à tous les objets agrégés par un même collecteur. Cela permet une sélection dynamique et non-anticipée des objets à soumettre à l'adaptation.

Le patron complet, résumé par les Figures 3.9, 3.10 et 3.6, représente des extensions légères de langage conformes à *Kernel-Lub*. Nous avons précisé l'impact de Lub et des Collecteurs sur la sémantique opérationnelle du langage hôte, qui décrit comment sont agrégés et adaptés les objets lors de l'exécution du programme.

Troisième partie

Mise en œuvre du patron de langage et application au déverminage non-anticipé

Cette partie contient trois chapitres. Les deux premiers chapitres présentent la mise en œuvre outillée du patron de langage présentée dans la deuxième partie de la thèse. Il s'agit de *Lub* pour l'adaptation non-anticipée d'objets et des *Collecteurs* pour la spécification non-anticipée des objets à adapter. Le dernier chapitre présente une infrastructure pour le débogage non-anticipé d'objets, basée sur *Lub* et sur les *Collecteurs*. Ces trois chapitres sont illustrés par des cas d'utilisation de déverminage en cours d'exécution, et présentent un positionnement des résultats de la thèse au regard de l'état de l'art présenté dans la première partie de la thèse.

Chapitre 4

Mise en oeuvre et évaluation du patron de langage pour l'adaptation de comportement non-anticipée : Lub

Sommaire

4.1 Principes de mise en œuvre de Lub dans un langage existant 108
4.1.1 Proposition de mise en œuvre 108
4.1.2 Avantages de la solution de mise en œuvre proposée 110
4.1.3 Inconvénients de la solution proposée 111
4.1.4 Mise en œuvre concrète au travers d'une *API* 113
4.2 Mise en œuvre dans Pharo avec Reflectivity 122
4.2.1 Reflectivity : la couche réflexive de Pharo 122
4.2.2 Une extension de Reflectivity pour une granularité objet . . . 124
4.2.3 Bénéfices de l'extension de Reflectivity 129
4.2.4 Modèle de mise en œuvre de Lub basé sur Reflectivity 130
4.2.5 Interfaces pour l'adaptation 132
4.3 Application du patron de langage : une mise en œuvre en Python avec les *Talents* 134
4.4 Analyse des performances 135
4.4.1 Critères de comparaison 135
4.4.2 Protocole de la prise de mesure 136
4.4.3 Périmètre des mesures 136
4.4.4 Évaluation de la vitesse du lookup instrumenté 137
4.4.5 Évaluation de la vitesse d'adaptation des objets 138
4.4.6 Évaluation de la consommation mémoire des objets adaptés . 140
4.4.7 Limites de l'évaluation des performances et de la consommation mémoire 142

4.5 Adaptation non-anticipée de comportement dans une flotte de drones 143
4.5.1 Scénario : la perte de GPS 144
4.5.2 Stratégie de déverminage : l'adaptation non-anticipée de comportement 145
4.5.3 Description de la simulation 145
4.5.4 Adaptations au fil de l'eau 146
4.5.5 Discussion 153
4.6 Évaluation 155
4.7 Conclusion 156

Lub est un patron de langage qui permet d'étendre un langage objet existant avec des capacités d'adaptation au niveau objet. Ce chapitre décrit la mise en œuvre de Lub et présente une évaluation de l'application du patron de langage à l'adaptation non-anticipée de comportement. La mise en œuvre de Lub est d'abord abordée de façon générale. Indépendamment du langage hôte choisi pour être étendu à l'adaptation dynamique de comportement, nous proposons une technique de mise en œuvre et nous en énonçons les principes généraux. Cette implémentation du patron de langage est incomplète, et notamment elle est limitée à l'adaptation de comportement des applications mono-processus. L'accumulation d'adaptations décrite dans le modèle d'exécution de Lub, ou la possibilité d'acquérir ou d'adapter les états d'un objet ne sont pas abordés dans cette mise en œuvre. Les adaptations gardées, bien qu'un choix d'implémentation soit exprimé, ne sont pas détaillées ni évaluées dans ce chapitre.

Une première mise en œuvre est proposée pour le langage Pharo, par le biais d'une extension à sa couche réflexive *Reflectivity*. L'extension de Reflectivity fournit un mécanisme d'adaptation d'objets sur lequel la mise en œuvre concrète de Lub pour Pharo peut reposer. Une seconde mise en œuvre est proposée pour le langage Python, suivant une implémentation *ad-hoc* des principes et techniques de mise en œuvre énoncés.

L'évaluation de Lub est effectuée en deux temps. Une première partie étudie l'impact de l'application de Lub sur les performances et la mémoire d'un code exécuté dans Pharo. Une seconde partie illustre et discute un cas d'application de l'adaptation non-anticipée de comportement au déverminage d'une simulation de drones. Dans cette simulation, l'échec d'un changement de contexte entraîne la perte du signal GPS d'un drone, et bloque la flotte entière. L'adaptation comportementale non-anticipée est mise en pratique pendant l'exécution du programme pour investiguer, comprendre et corriger le problème. Enfin, nous évaluons la mise en œuvre de Lub dans Pharo au regard des propriétés que nous avons identifiées pour l'adaptation non-anticipée de comportement.

Ces travaux sur l'extension de langage avec des capacités d'adaptation non-anticipée de comportement ont donné lieu à deux publications [Costiou et al., 2016, Costiou et al., 2018a].

4.1 Principes de mise en œuvre de Lub dans un langage existant

Nous définissons des principes de mise en œuvre de *Lub*. Ce sont des représentations logicielles abstraites, basées sur *Kernel-Lub*. Ces principes constituent des guides de haut niveau pour décrire une mise en œuvre possible, pouvant être appliquée à tous les langages éligibles à *Lub*. Pour être éligible à *Lub*, un langage doit être conforme à *Kernel-Lub*, ou tout du moins doit pouvoir être représenté par une extension de *Kernel-Lub*. Pour chaque brique logicielle abstraite, des contraintes de mise en œuvre sont énumérées. Ce sont concrètement des besoins au niveau langage, pour que ce dernier soit éligible pour être étendu par les solutions avancées dans cette thèse.

4.1.1 Proposition de mise en œuvre

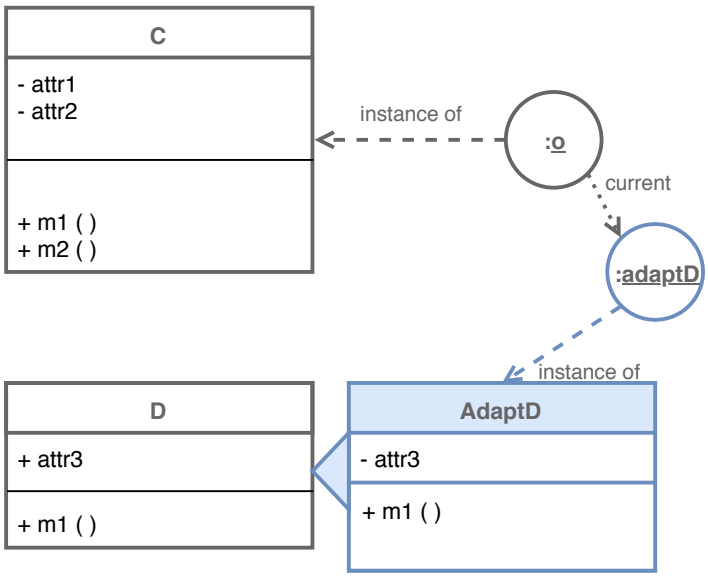


FIGURE 4.1 – Un objet *o*, instance de la classe *C*, est adapté à l'exécution par *adaptD*, instance de l'adaptation *AdaptD*.

La Figure 4.1 présente un objet *o*, instance d'une classe *C* adapté par *adaptD*, une instance de l'adaptation *AdaptD*. Cet exemple décrit un modèle conforme au modèle d'exécution de *Lub* (Figure 3.6). Ce modèle peut concrètement être mis en œuvre par la migration de l'objet vers une sous-classe anonyme de sa propre classe, illustrée par la Figure 4.2.

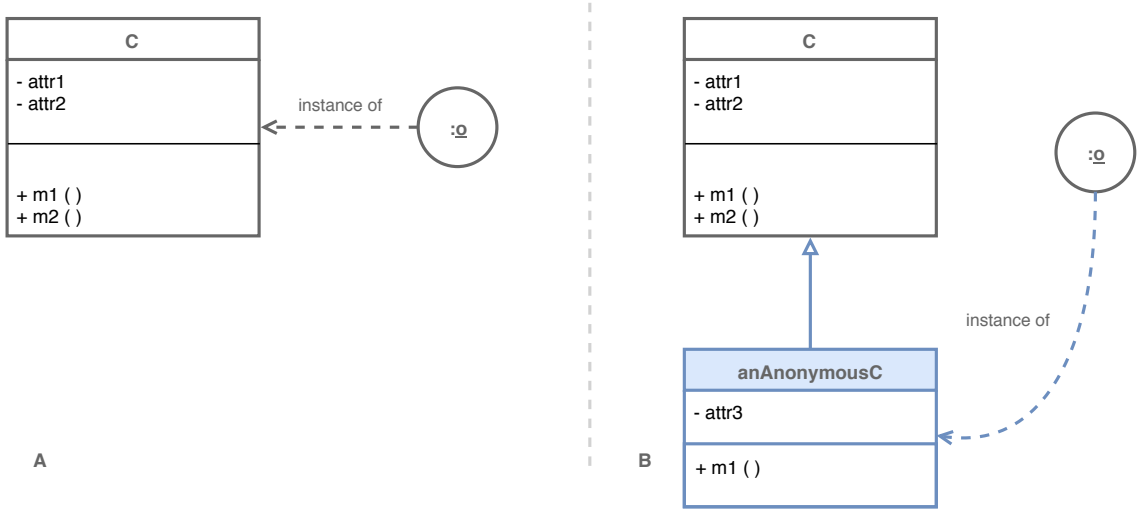


FIGURE 4.2 – Adaptation d'un objet. **A** : l'objet *o* instance de la classe *C* est dans un état non adapté. **B** : l'objet *o* a été adapté. Il est désormais instance d'une classe anonyme, sous-classe de sa classe d'origine *C*.

Lors de l'adaptation, l'objet *o* est migré d'un état non adapté (étape A de la Figure 4.2) vers une sous-classe anonyme de sa propre classe. Dans son état adapté, il est donc instance d'une nouvelle sous-classe de *C* (étape B de la Figure 4.2). Cette sous-classe est dite *anonyme* : elle n'a pas de nom et n'a pas d'existence à proprement parler d'un point de vue externe au système. Elle n'est pas développée manuellement, mais générée à partir des spécifications d'une adaptation. Elle n'est pas non plus, idéalement, visible du point de vue de base des outils du langage car l'objet n'est pas supposé avoir changé d'identité. Elle peut éventuellement être totalement visible au niveau méta, si l'outillage du langage le permet. En dehors de ces spécificités, la sous-classe anonyme possède exactement les mêmes caractéristiques que les autres classes du langage : elle est unique et ne peut hériter que d'une seule super classe.

Contrainte de mise en œuvre n^{o}1

Le langage hôte doit permettre de définir et de charger de nouvelles classes lors de l'exécution d'un programme.

L'objet en cours d'adaptation est ensuite *migré* vers sa nouvelle classe. La migration implique le changement d'interface et le transfert d'état dans la nouvelle structure de l'objet. *Lub* ne permet pas la suppression de comportements ou

d'états. Méthodes comme attributs ne peuvent donc pas être perdus après une adaptation. Une méthode peut être acquise ou modifiée par une adaptation – dans ce cas uniquement son corps est modifié mais pas sa signature. Un attribut ne peut qu'être acquis lors d'une adaptation, mais l'opération peut nécessiter de migrer l'état des attributs dans la structure d'origine de l'objet vers sa nouvelle structure.

Contrainte de mise en œuvre n^{o}2

Le langage hôte doit fournir des primitives pour la migration d'objets distincts, de leur classe d'origine vers une autre classe, lors de l'exécution d'un programme.

4.1.2 Avantages de la solution de mise en œuvre proposée

La migration d'un objet vers une sous-classe directe de sa classe d'origine, qui implémente les variations comportementales spécifiées par l'adaptation, présente plusieurs avantages.

La sémantique du lookup de *Lub* est traduite immédiatement, sans instrumentation particulière de ce dernier. Le mécanisme standard du lookup sera appliqué lors d'un envoi de message, et commencera dans la classe de l'objet, c'est-à-dire la sous-classe anonyme. Comme cette dernière n'implémente que les variations comportementales imposées par l'adaptation appliquée, le lookup sera résolu dans cette sous-classe pour ces variations, et continuera de manière standard dans sa super classe – c'est-à-dire la classe d'origine de l'objet – pour toutes les méthodes et états non adaptés. Le fait de reposer sur le mécanisme standard du lookup lors de l'exécution limite également un surcoût éventuel en terme de performances, par rapport à une instrumentation dynamique du lookup [Ducasse, 1999].

Comme l'objet a migré vers une sous-classe directe de sa classe d'origine, son identité au sens du *self* est préservée. Comme *self* référence toujours, de manière invariable, le receveur du message, le mécanisme de base du lookup s'appliquera pour tous les envois de message à *self* à partir de la sous-classe anonyme (donc comprenant les éventuelles méthodes adaptées). Le problème du *self* [Lieberman, 1986, Ducasse, 1999] est évité sans traitement particulier.

La compatibilité avec les outils du langage est maintenue, car la mise en œuvre repose sur des constructions standards du langage. Il n'y a donc pas de traitements spécifiques à intégrer pour que l'objet adapté soit reconnu par l'outillage, par exemple un débogueur, ou par des fonctionnalités d'introspection.

4.1.3 Inconvénients de la solution proposée

Compiler une adaptation dans une sous-classe anonyme présente plusieurs problèmes de mise en œuvre. Ces problèmes sont principalement dus à la compilation de méthodes venant d'autres classes, et devenant partie du comportement de l'objet adapté.

Problème du *super* Lorsqu'un message est envoyé à la variable spéciale *super*, le lookup pour ce message doit démarrer dans la super classe de la classe de l'objet receveur du message (désigné par *super*). Ce comportement est également attendu dans le cas d'une méthode adaptée ou acquise - et donc compilée dans une sous-classe anonyme de la classe d'origine du receveur. Ce cas de figure est illustré par la Figure 4.3, où la méthode *method()* dans la classe *aClass* fait explicitement référence à la méthode *method()* de *aSuperClass*, super classe de *aClass*, par le biais d'un envoi de message à *super*. La version adaptée de cette méthode est compilée dans une sous-classe anonyme de *aClass*, nommée *anAnonymousClass*. Mais seule une partie de son comportement est adapté, et l'envoi de message à *super* n'est pas altéré. Du point de vue de l'adaptation, le lookup doit retrouver et exécuter la méthode présente dans la super classe d'origine de *aClass*. Du point de vue du système, la super classe directe de *anAnonymousClass* est bien la classe d'origine, et le lookup sera résolu dans *aClass* au lieu de *aSuperClass*.

Ce problème, dû à la mise en œuvre décrite dans la section 4.1.1, est susceptible de provoquer deux exécutions de la même méthode (la version adaptée et la version d'origine), ou bien l'exécution de la mauvaise version d'une méthode (celle de la classe d'origine au lieu de la *super méthode*). Il est également très probable que ce lookup erroné mette les objets dans des états incohérents, les méthodes concernées pouvant non seulement modifier les attributs de l'objet courant, mais également interagir avec d'autres objets. Dans les cas les plus sensibles, cela peut amener à l'interruption de l'exécution du programme.

Il est nécessaire de prendre en compte ce cas de figure lors de la mise en œuvre par la technique des sous-classes anonymes. Plusieurs possibilités sont offertes, par exemple pour une méthode adaptée appelant *super* il est possible d'aplatir tous les appels à *super* de la hiérarchie de super classes. Cela revient à compiler directement une copie de la méthode recherchée dans la sous-classe anonyme pour rediriger son lookup vers *self*. Il faudra alors s'assurer que cette nouvelle méthode ne contient pas elle-même des appels à *super*, auquel cas il sera nécessaire de répéter l'opération. L'impact en terme de mise en œuvre est alors plus subtil et plus complexe qu'une compilation des méthodes adaptées dans une sous-classe. Une autre solution consisterait à reposer sur un mécanisme d'adaptation pré-existant capable de gérer ce cas de figure, ou qui pourrait en faciliter la mise en œuvre.

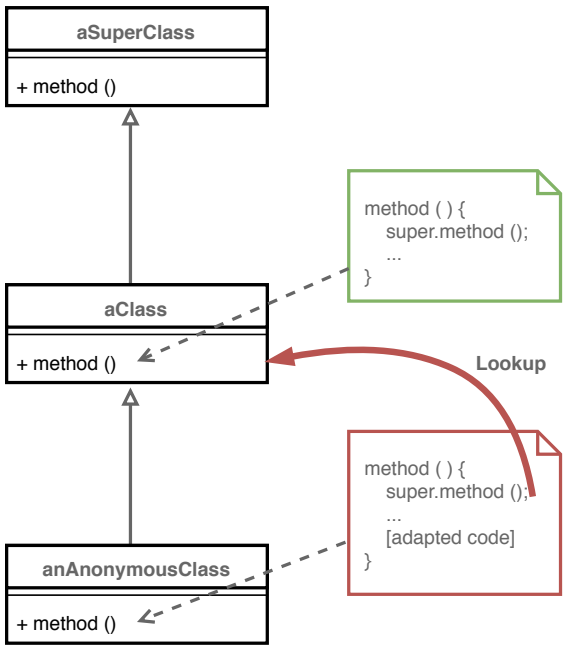


FIGURE 4.3 – Problème du *super*, dû à la mise en œuvre par sous-classes anonymes. Le lookup pour les envois de message à la pseudo-variable super est démarré dans la classe d'origine du receveur (*aClass*) au lieu de démarrer dans la super classe de cette dernière (*aSuperClass*).

Erreurs liées à l'adaptation Bien que Lub garantisse la non réduction d'interface d'un objet adapté, l'adaptation consiste à étendre l'interface de cet objet et il est possible que des échecs du lookup surviennent dans le corps des méthodes adaptées. Le principe de mise en œuvre énoncé propose de compiler les méthodes d'extension dans une sous-classe de l'objet, mais ne spécifie pas comment détecter et traiter les erreurs sémantiques du code adapté. Par exemple une méthode adaptée pourrait, par chaînage d'appels de méthode, effectuer des accès à des attributs non existants. De la même manière, des méthodes non implémentées peuvent être appelées, et pour lesquelles le lookup échouera. Ces deux exemples ne sont pas forcément visibles immédiatement dans le code d'une méthode adaptée. Le problème est indépendant de Lub, et se pose également dans le cas d'un développement classique. La difficulté est qu'il faudrait non seulement repérer les erreurs immédiates dans une méthode à sa compilation, mais également s'assurer que chaque appel, chaque accès dans le corps des méthodes appelées aboutira à un succès du lookup. L'analyse statique du code à la compilation est rendue difficile à l'exécution dans les langages dynamiquement typés tels que ceux pouvant être décrits par *Kernel-Lub*. Les principes d'implémentation énoncés ne suffisent pas pour détecter ces cas d'erreurs, et devraient être

complétés par des mécanismes supplémentaires (par exemple de l'analyse statique pré-compilation) pour garantir une sécurité totale de l'adaptation. Il est intéressant de noter que ce problème n'est pas inhérent à Lub, mais apparait dans des cas de mise à jour dynamique du programme par des moyens plus conventionnels. C'est par exemple un problème classique du *Live Programming* [Mattis et al., 2017].

Donc le lookup n'échouera jamais lors d'un appel à une méthode adaptée, en revanche le corps de cette méthode, c'est-à-dire le code adapté par l'utilisateur-rice, peut contenir des appels qui provoqueront des échecs du lookup. Ce problème, dû au code écrit par l'utilisateur-rice, est indépendant de Lub et sort du cadre de la thèse.

Coût de l'adaptation lors de l'exécution Migrer un objet vers une sous-classe anonyme contenant le comportement adapté implique de compiler cette classe. Lorsque cette opération est effectuée pendant l'exécution du programme, il y a un coût en terme de performances (temps de compilation) et de mémoire (nouvelles classes ajoutées au système). Il serait possible de minimiser ce coût en mettant en œuvre des techniques de compilation *hors ligne*, c'est-à-dire à partir d'un ordinateur distant, puis en mettant à jour le code du programme à distance. Cependant l'aspect non-anticipé de l'adaptation ne garantit pas qu'on puisse connaître la classe de l'objet en avance afin de produire sa sous-classe anonyme. Premièrement, il faudrait obtenir une référence à l'objet avant son adaptation pour connaître sa classe. Puis, il faudrait compiler son adaptation *hors ligne*, la charger dans le programme en cours d'exécution et adapter l'objet. Il n'y a pas de garantie que le contexte de l'adaptation soit encore valide à ce moment, ou que l'objet soit encore présent en mémoire. Ce problème est orthogonal à la mise en œuvre proposée, qui ne décrit pas comment l'adaptation est appliquée à l'exécution. Pour une mise en pratique réelle d'une telle adaptation, il y a nécessité de reposer sur un mécanisme de communication et d'interaction avec le programme, et dans un sens, de mise à jour dynamique de ce dernier (injection de méthodes, ordres de migration d'objets). Dans cette thèse, ces problématiques ne sont pas abordées, et la mise en œuvre repose sur une version *naïve* des principes d'implémentation généraux proposés ne tenant pas compte de toutes les optimisations possibles en terme de compilation ou d'application de l'adaptation.

4.1.4 Mise en œuvre concrète au travers d'une *API*

Nous proposons une mise en œuvre concrète de Lub suivant les principes précédemment énoncés sous la forme d'une *API*. Cette *API* est fournie par un adaptateur, qui prend en charge l'adaptation des objets à partir d'un couple $\{objet, adaptation\}$. Ce dernier repose sur un mécanisme d'adaptation mis en oeuvre dans le langage hôte. Le mécanisme d'adaptation peut être construit à partir des contraintes 1 et 2 en

suivant les principes proposés dans la suite de ce chapitre. Il peut aussi être un mécanisme sous la forme d'une bibliothèque logicielle mettant en oeuvre au moins les contraintes 1 et 2 si elles ne sont pas satisfaites nativement par le langage. La bibliothèque peut également apporter des fonctionnalités supplémentaires non spécifiées et non supportées par *Lub*, par exemple des optimisations de performances ou d'utilisation mémoire, des aspects temps-réel ou parallèles, etc. Ces aspects et les apports potentiels d'un mécanisme d'adaptation spécifique ne sont pas discutés dans cette thèse.

L'adaptateur et son *API* sont décrits dans la section 4.1.4.1, tandis que le mécanisme d'adaptation sous-jacent est décrit dans les sections 4.1.4.2 et 4.1.4.3.

4.1.4.1 L'adaptateur d'objets

L'adaptateur est une interface pour contrôler le processus d'adaptation d'un objet. Il est décrit par la Figure 4.4 par la classe *Adaptateur*. L'adaptateur interagit avec un mécanisme d'adaptation qui peut être *ad-hoc*, ou une bibliothèque existante du langage hôte. Ce mécanisme doit satisfaire au minimum aux contraintes 1 et 2 énoncées dans la section 4.1.1. Dans le cas où le langage hôte ne fournirait pas ce mécanisme d'adaptation, nous décrivons une mise en œuvre possible en deux parties, un gestionnaire de sous-classes anonymes (décrit dans la section 4.1.4.2) et un migrateur d'objet (décrit dans la section 4.1.4.3).

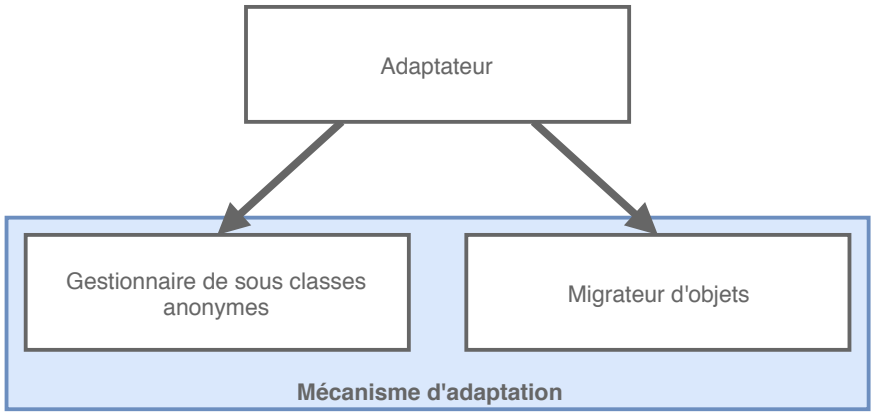


FIGURE 4.4 – L'adaptateur est une interface de contrôle du mécanisme d'adaptation, divisée en deux blocs : un gestionnaire de sous-classes anonymes et un migrateur d'objets.

L'*API* fournie par l'adaptateur permet au développeur ou à la développeuse d'appliquer une adaptation à un objet. L'adaptateur repose sur son mécanisme d'adaptation pour *1-* définir une sous-classe anonyme de la classe de l'objet qui

implémente le comportement adapté, et *2-* migrer l'objet de sa classe originale vers sa classe fille nouvellement créée. Ce sont les deux étapes principales de l'adaptation d'un objet, décrite par la Figure 4.5. La méthode *adapt*, prenant en paramètre l'objet cible et son adaptation, déclenche le processus d'adaptation et retourne l'objet adapté.

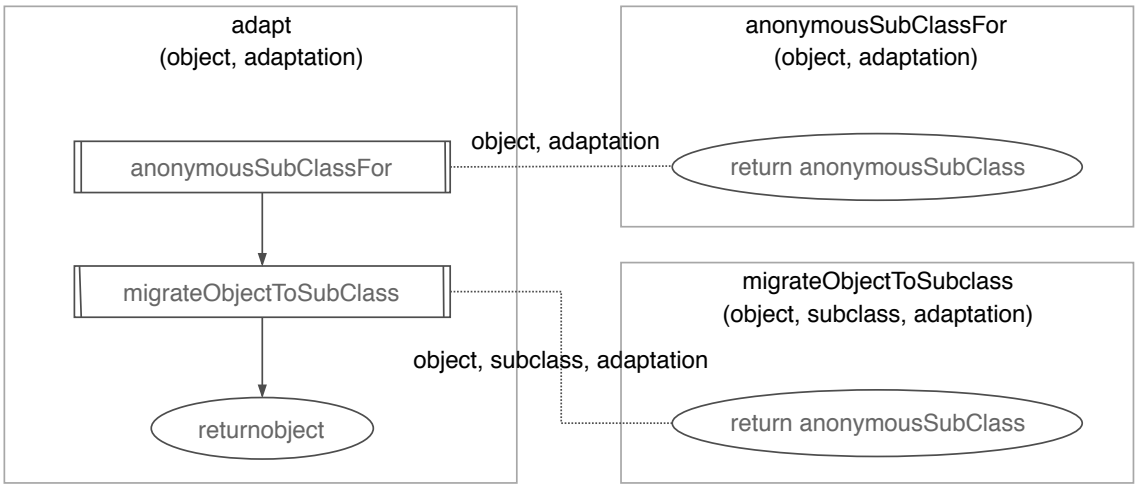


FIGURE 4.5 – L'interface *adapt* de l'adaptateur prend en paramètre un objet et une adaptation : une sous-classe anonyme de l'objet est générée avec la modification de comportement spécifiée par l'adaptation, puis l'objet est migré vers cette nouvelle classe. Les fonctions appelées *anonymousSubClassFor* et *migrateObjectToAnonymousSubclass* sont des interfaces respectivement du gestionnaire de sous-classes anonymes et du migrateur d'objet.

L'opération inverse est appliquée en demandant l'adaptation d'un objet avec le littéral *null* en guise d'adaptation. L'interface utilisée reste la même, et l'objet est simplement migré de sa classe adaptée vers sa super classe, c'est-à-dire la classe d'origine de l'objet. Pour des raisons de simplicité, cette opération n'est pas décrite dans l'algorithme de la Figure 4.5. En effet, l'annulation d'une adaptation sur un objet est une opération moins complexe qui ne génère pas de sous-classe anonyme et se contente de migrer l'objet vers sa classe d'origine.

Il n'y a pas de contrainte sur la forme de l'adaptateur, qu'il soit instancié ou utilisé en tant que variable globale (par exemple avec des méthodes statiques dans la classe *Adaptateur*) ne dépend que des facilités du langage. La possibilité de décliner l'interface globale *adapt* de l'adaptateur, ou de la mise en œuvre de sucre syntaxique

pour les opérations d'adaptation et/ou la définition d'adaptations, est laissée aux besoins de l'utilisateur-rice en fonction du langage hôte.

4.1.4.2 Le gestionnaire de sous-classes anonymes

Le gestionnaire de sous-classes anonymes, décrit par la Figure 4.6, est la première brique du mécanisme d'adaptation utilisé par l'adaptateur d'objets. Son rôle est de produire, à partir d'un objet et d'une instance d'*Adaptation* une sous-classe directe de la classe d'origine de l'objet. Cette sous-classe est compilée avec :

— De nouvelles méthodes et attributs qui sont des comportements et états acquis par l'adaptation.
— Des méthodes qui redéfinissent celles de la classe d'origine, et qui modifient le comportement de base de l'objet.

Il s'agit d'une automatisation de la spécialisation de comportement par héritage direct, ce que fournissent la plupart des langages objets y compris ceux décrits par *Kernel-Lub*.

Les opérations de génération de classes nécessitent de la compilation de code, ce qui peut poser problème lors d'une instrumentation massive d'un système en cours d'exécution. C'est le cas dans la mise en œuvre proposée, où nous choisissons d'effectuer ces opérations directement sur la cible – c'est-à-dire le programme en cours d'exécution. Le gestionnaire peut optimiser légèrement cette situation, en conservant en mémoire les sous-classes déjà générées pour un couple $\{objet, adaptation\}$ (Figure 4.6). Lors d'une requête de l'adaptateur pour obtenir une nouvelle classe, le gestionnaire vérifie d'abord si une classe implémentant exactement l'adaptation demandée existe déjà et le cas échéant la renvoie directement à l'adaptateur. Une même adaptation n'est ainsi jamais compilée deux fois dans une classe anonyme pour un même type d'objet. Cette optimisation est efficace lorsqu'une même adaptation est demandée de nombreuses fois pour un même type d'objet. Elle perd toute son efficacité lorsque de nombreux types d'objets différents doivent être adaptés, ou qu'ils doivent être adaptés par des adaptations différentes.

L'algorithme général du gestionnaire de sous-classes anonymes est décrit par la Figure 4.7. L'interface publique, utilisée par l'adaptateur d'objet, est la méthode *anonymousSubClassFor*. Cette dernière prend en entrée un couple $\{objet, adaptation\}$ pour générer la classe adaptée. Une sous-classe est toujours générée à partir de la classe d'origine de l'objet, et non pas de sa classe courante. Un objet peut en effet être adapté à l'instant courant, et donc avoir déjà été migré vers une classe anonyme. Dans ce cas, le gestionnaire doit prendre soin de partir de la classe originelle de l'objet pour créer la classe adaptée, ou de retrouver une classe correspondante précédemment générée et stockée dans la couche d'optimisa-

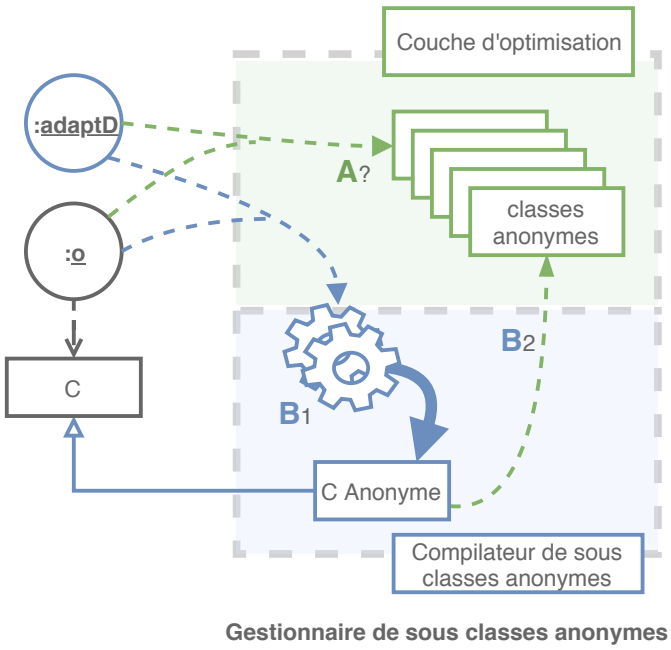


FIGURE 4.6 – Le gestionnaire prend en entrée une adaptation et un objet pour générer une classe anonyme. Si une classe correspondant exactement à l'adaptation pour l'objet spécifié a déjà été générée, elle est retournée immédiatement (A). Sinon, la sous-classe est générée à partir de la classe de l'objet, puis les modifications de comportement spécifiées par l'adaptation sont compilées dans cette sous-classe (B_1). Enfin, la sous-classe est stockée par la couche d'optimisation (B_2) avant d'être retournée.

tion. Pour des raisons de simplicité, cette subtilité n'apparaît pas dans l'algorithme présenté.

Les opérations de compilation du comportement adapté doivent prendre en compte les éventuels appels à la variable *super*, afin d'éviter le problème du *super* (section 4.1.3). Dans le code adapté, *super* doit toujours faire référence à la super classe de la classe d'origine de l'objet, et non pas à la super classe de sa classe courante (qui est une classe fille de la classe d'origine).

Les inconvénients de cette mise en œuvre, décrits à la section 4.1.3, pourraient être résolus lors de la compilation. Idéalement, cette compilation devrait être effectuée *hors-ligne*, afin d'éviter des ralentissements du programme en cours d'exécution. Une analyse poussée des méthodes adaptées pourrait être conduite pour garantir que tous les appels directs ou indirects issus de l'exécution du code adapté peuvent être

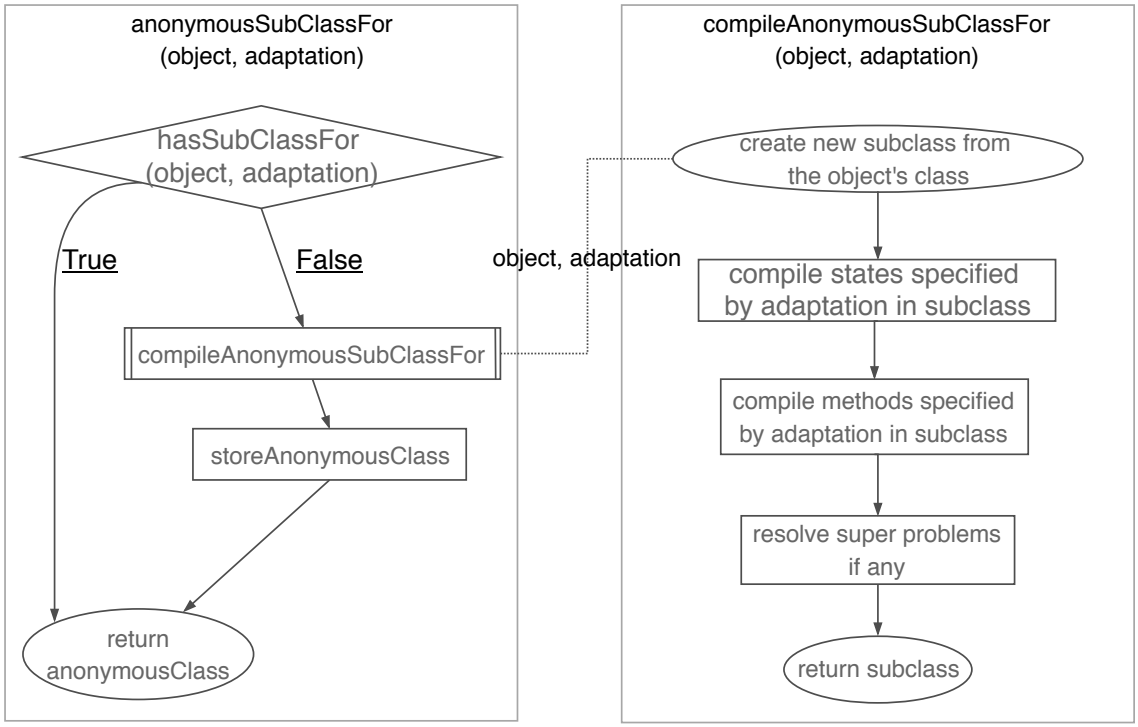


FIGURE 4.7 – Algorithme de génération de sous-classe anonyme pour un couple $\{objet, adaptation\}$. L'interface publique pour demander la génération d'une classe est la méthode publique *anonymousSubClassFor*.

résolus. Ces aspects ne sont pas pris en compte dans la mise en œuvre courante, et ne sont donc pas décrits dans les modèles et les algorithmes présentés. Il conviendra cependant de les considérer pour une mise en œuvre moins naïve destinée à une expérimentation dans des conditions plus critiques, par exemple une application industrielle.

4.1.4.3 Le migrateur d'objets

Le migrateur d'objet est la seconde brique du mécanisme d'adaptation utilisé par l'adaptateur d'objets. Son rôle est de migrer des objets de leur classe courante vers une classe anonyme. Cette dernière est une classe fille de la classe d'origine de l'objet, et met en œuvre une adaptation de comportement pour l'objet ciblé. La migration pour un objet o, instance de la classe C, vers la classe C *Anonyme*, sous-classe de C, est illustrée par la Figure 4.8. L'objet peut éventuellement être déjà adapté au moment d'une migration, c'est-à-dire qu'il a précédemment été migré vers une sous-classe anonyme. L'opération de migration vers une sous-classe différente, par

construction, change l'adaptation courante – qui est remplacée par celle implémentée par la nouvelle sous-classe anonyme.

La migration d'une classe vers une autre doit prendre en compte la structure courante de l'objet, et s'assurer que son état est correctement transféré vers sa nouvelle structure. Le changement de structure est limité par le fait que la nouvelle classe hérite directement de la classe d'origine de l'objet. La définition structurelle de la classe d'origine est commune entre la classe mère et sa fille, qui partagent un ensemble de méthodes et d'attributs. La différence structurelle se situera alors au niveau des attributs et méthodes acquis via l'adaptation. Le coût de migration peut alors être limité, selon les possibilités du langage hôte, si l'adaptation est uniquement comportementale et qu'il n'y a pas d'états à migrer.

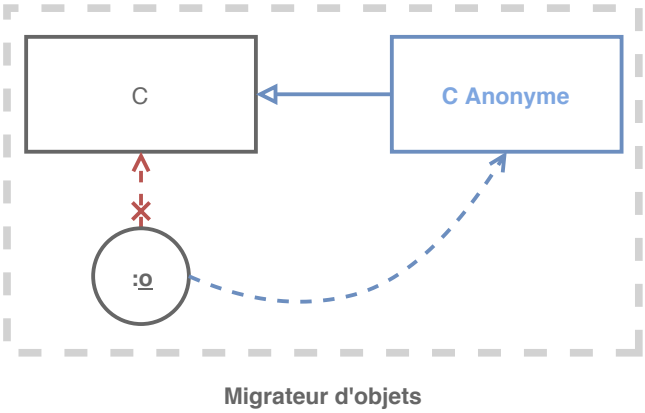


FIGURE 4.8 – À partir d'un couple {*objet*, *classe anonyme*}, le migrateur brise le lien entre l'objet et sa classe d'origine, et lie l'objet à la classe anonyme qui devient sa classe courante. Les états de l'objet sont migrés vers sa nouvelle structure, et les nouveaux états éventuels de l'objet sont initialisés.

L'algorithme de la Figure 4.9 décrit les opérations de migration ou de réversion d'adaptation effectuées par le migrateur d'objets. Deux méthodes constituent l'interface utilisable du migrateur :

— *migrateObjectToAnonymousSubClass* : prend en paramètre l'objet à migrer, la classe cible pour la migration et l'adaptation. Nous supposons ici l'existence d'une primitive de migration d'un objet de sa classe vers une autre classe (contrainte $n^o 2$, section 4.1.1). En guise d'illustration, nous utilisons la primitive *become* des langages basés sur Smalltalk [Goldberg and Robson, 1983], qui migre l'objet de sa structure d'origine vers une nouvelle structure. L'algorithme doit s'assurer, une fois la migration structurelle effectuée, que les

éventuels nouveaux attributs de l'objet sont correctement initialisés, suivant la spécification fournie par l'adaptation.

— *revertObjectAdaptation* : prend en paramètre l'objet à restaurer. La réversion consiste à migrer un objet couramment adapté vers sa classe d'origine, afin d'annuler l'adaptation. Cette méthode est appelée par l'adaptateur d'objet lorsque le littéral *null* lui est passé en guise d'adaptation. Cette subtilité n'est pas précisée dans l'algorithme général de l'adaptateur d'objet (section 4.1.4.1) pour des raisons de simplicité.

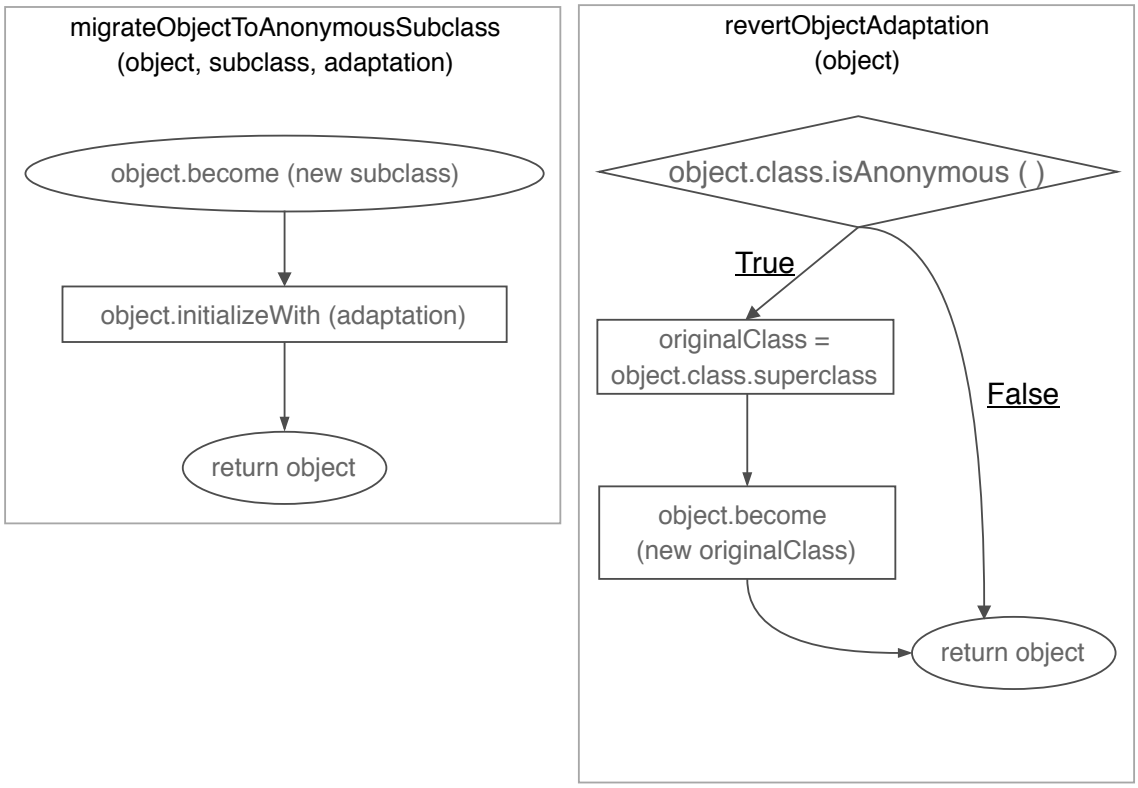


FIGURE 4.9 – Algorithmes généraux de l'interface de migration : les méthodes *migrateObjectToAnonymousSubClass* et *migrateObjectToAnonymousSubClass*.

L'utilisation de la primitive *become* dans l'algorithme de la Figure 4.9 permet de rendre l'opération de migration transparente du point de vue du système. Si dans cette illustration (Figure 4.9) une nouvelle instance est créée pour migrer l'objet vers sa nouvelle structure, la primitive doit s'assurer que :

— L'état de l'objet est correctement migré vers celui du nouvel objet.

— Les références à l'ancienne structure de l'objet par les autres entités du système sont mises à jour pour faire référence à la nouvelle.

Cette primitive peut prendre différentes formes, comme le *become*: en Smalltalk, ou le changement de pointeur du méta-objet – c'est-à-dire la classe de l'objet – directement dans l'objet en Python. D'autres formes sont possibles, selon ce que fournit le langage hôte et dans le respect de la contrainte $n^o 2$ (section 4.1.1).

4.1.4.4 Les adaptations gardées

La mise en œuvre proposée restreint légèrement la sémantique opérationnelle informelle décrite dans la section 3.2. La sémantique opérationnelle spécifie que la garde d'une adaptation est évaluée à chaque appel de méthode afin de déterminer si, selon la mise en œuvre décrite précédemment, le point de départ du lookup commence dans la sous-classe anonyme ou dans la classe d'origine de l'objet adapté. Pour ce faire, un code évaluant la garde peut être généré lors de la compilation des méthodes de l'adaptation dans la sous-classe anonyme. Ce code peut être placé en tant que garde en entête de chaque méthode adaptée compilée dans la sous-classe. Si le résultat de son évaluation n'est pas le littéral *true*, alors le lookup est redémarré dans la classe d'origine de l'objet par un appel à *super*. Bien que techniquement réalisable sans difficulté particulière, ce choix pose plusieurs problèmes :

— les temps de compilation sont rallongés par la génération de la garde,
— le changement de garde dynamique implique la régénération de la sous-classe anonyme, ce qui peut ralentir considérablement l'exécution du programme,
— à l'exécution, un test supplémentaire est exécuté pour chaque appel de méthode adaptée, ce qui peut ralentir considérablement le temps d'exécution et notamment en fonction de la complexité de la garde.

Une implémentation partielle est proposée pour pallier ces défauts. L'évaluation de la garde est intégrée au moment de la tentative d'adaptation d'un objet dans la méthode *adapt* de l'adaptateur d'objets. À chaque fois qu'une demande d'adaptation est effectuée pour un objet, la garde est évaluée et l'objet n'est adapté que si le résultat de cette dernière est le littéral *true*. Cette mise en œuvre n'est pas tout à fait conforme à la sémantique opérationnelle de l'adaptation gardée, qui spécifie des gardes d'adaptation beaucoup plus fines (à chaque appel de méthode). Le choix de conditionner l'adaptation complète au résultat de l'évaluation de la garde permet d'économiser sur les ralentissements énumérés ci-dessus, mais fait perdre en finesse sur la condition d'activation d'une adaptation. En conséquence de ce choix, un objet n'est adapté que si sa garde est évaluée à *true* et tous les appels de méthodes ultérieurs considèrent que la garde est toujours évaluée à *true*.

4.2 Mise en œuvre dans Pharo avec Reflectivity

Reflectivity [Denker, 2008] est une bibliothèque intégrée à Pharo, et constitue une partie de la couche réflexive du langage. Reflectivity permet de mettre en œuvre une réflexivité structurelle et comportementale au niveau d'une méthode et des sous-éléments qui la composent − par exemple au niveau d'une expression. Les outils de méta-instrumentation fournis par Reflectivity permettent la mise en œuvre de variations comportementales, et ont l'avantage d'être nativement intégrées à Pharo. Cependant, ces méta-instrumentations ne se font qu'au niveau des classes. Il n'est donc pas possible d'instrumenter des objets spécifiques. Si Reflectivity représente une base intéressante pour jouer le rôle de mécanisme d'adaptation utilisé par l'adaptateur d'objet (section 4.1.4.1), ce n'est pas suffisant pour obtenir les capacités d'adaptation décrites par Lub.

Dans cette section, nous présentons rapidement Reflectivity et ses limitations pour la mise en œuvre d'adaptations comportementales. Nous décrivons ensuite une extension de Reflectivity pour obtenir un grain d'adaptation au niveau objet. Puis, nous présentons une mise en œuvre de Lub basée sur l'extension de Reflectivity. Nous précisons l'utilisation de l'extension de Reflectivity pour l'adaptation au sens Lub, et notamment le modèle et les interfaces de contrôle de l'adaptation. Cette implémentation ne prend pas en compte l'adaptation de l'état d'un objet, et se focalise sur l'adaptation de comportement (méthodes).

4.2.1 Reflectivity : la couche réflexive de Pharo

4.2.1.1 MetaLink

Les metalinks sont des annotations de l'arbre de syntaxe abstraite de Pharo (AST). Ce sont, dans le modèle de *Kernel-Lub*, des annotations ou des instrumentations d'expressions (Figure 3.2). Lorsque l'expression est évaluée pendant l'exécution du programme, l'annotation déclenche l'exécution d'un comportement spécifique.

Un metalink (Figure 4.10) se paramètre avec (1) un *méta-objet*, qui implémente le comportement à exécuter, (2) un *sélecteur* avec sa liste éventuelle d'arguments qui sera envoyée au méta-objet pour déclencher l'exécution du comportement et (3) un *contrôle*, qui définit comment ce comportement s'exécute, au regard du nœud annoté par le metalink dans l'arbre de syntaxe abstraite. Le contrôle définit si l'annotation s'exécute avant, après ou à la place de l'expression annotée. La Figure 4.10 illustre l'utilisation d'un metalink. Le méta-objet est un bloc qui affiche une chaîne de caractères dans la console (ligne 2), et le sélecteur correspond au symbole *#value* (ligne 3). Lors de l'exécution du metalink, le message *#value* sera envoyé au méta-objet (le bloc). Le contrôle définit est *#instead* (ligne 4), ce qui aura pour effet de remplacer l'exécution de l'expression ciblée par l'exécution du metalink. Enfin, un nœud de la

syntaxe abstraite est annoté par le metalink (ligne 5) : il s'agit du nœud représentant la méthode *#add*: de la classe *OrderedCollection*. Cette opération d'annotation a pour effet de recompiler la méthode contenant le nœud ciblé, avec l'instrumentation apportée par le metalink. Dans cet exemple, lors de l'exécution du programme tous les envois du message *#add*: aux instances de *OrderedCollection* seront remplacés par l'impression du message *'Hello World'* dans la console. Le metalink peut être installé et désinstallé dynamiquement lors de l'exécution du programme. En outre, les instrumentations apportées par les metalinks sont dites "*non-intrusives*". Les annotations de l'AST ne provoquent pas d'insertion de code dans le programme, et les méthodes recompilées avec des metalinks existent en parallèle des méthodes originales.

```
1 metalink := MetaLink new.
2 metalink metaObject: [Transcript show: 'Hello World'].
3 metalink selector: #value.
4 metalink control: #instead.
5 (OrderedCollection lookupSelector: #add:) ast link: metalink.
```

FIGURE 4.10 – Création et installation d'un metalink sur le noeud de l'arbre de syntaxe abstraite représentant la méthode *add*: de la classe *OrderedCollection*. À l'exécution, tous les envois du message *#add*: aux instances de *OrderedCollection* seront remplacés par l'impression du message *'Hello World'* dans la console.

4.2.1.2 Limitations

L'installation d'un metalink ne peut se faire qu'au niveau d'une classe. Toutes les instances d'une classe possédant une méthode instrumentée par un metalink sont donc affectées par cette instrumentation. Une mise en œuvre de Lub utilisant Reflectivity comme mécanisme d'adaptation nécessiterait de pouvoir installer des metalinks sur des objets distincts. Il est tout de même possible d'ajouter des conditions aux metalinks. Un filtre peut être mis en place pour sélectionner pour quel objet le metalink s'exécutera. Cela a deux effets collatéraux :

1. Tous les objets de la classe possédant une méthode instrumentée seront dans les faits impactés. Tous ces objets exécuteront la condition du metalink pour déterminer s'ils doivent l'exécuter ou pas. Une altération de comportement est donc bien appliquée sur tous les objets de cette classe.
2. L'exécution de la condition pour filtrer l'objet auquel appliquer le metalink est source de ralentissement potentiellement considérable pour le programme en cours d'exécution. La seule raison de cet écueil est ici l'application du metalink à tous les objets de la classe avec une méthode instrumentée, avec

une condition à évaluer pour savoir si le metalink doit s'exécuter ou pas. Cela n'aurait pas lieu d'être si le metalink était uniquement appliqué à l'objet ciblé par la condition.

Reflectivity possède d'autres limitations, non étudiées dans le cadre de cette thèse, et notamment :

— Les méthodes déjà présentes sur la pile d'exécution ne peuvent pas être instrumentées. Seuls les nouveaux appels à cette méthode pourront exécuter le metalink.
— De manière corollaire au point précédent, les expressions dans une boucle en cours d'exécution (c'est-à-dire sur la pile) ne peuvent pas être annotées par un metalink tant que la boucle n'est pas terminée. Les boucles infinies en cours d'exécution ne peuvent donc jamais être instrumentées.
— L'exécution des méthodes instrumentées est par construction compatible avec les programmes multi-processus. En revanche, l'installation d'un metalink ne l'est pas. L'utilisation de Reflectivity est donc limitée aux environnements mono-processus.

Dans la mise en œuvre présentée dans cette thèse, nous considérons que nous héritons de ces dernières limitations. L'étude et la résolution de ces problèmes se situent en effet en dehors du cadre de la thèse.

4.2.2 Une extension de Reflectivity pour une granularité objet

Pour obtenir des instrumentations à granularité objet basées sur les metalinks, nous étendons Reflectivity avec un gestionnaire de sous-classes anonymes − tel que décrit à la section 4.1.4.2 − et un gestionnaire de metalink centré objet. L'objectif est d'obtenir un mécanisme d'adaptation d'objets complètement intégré au langage sur lequel puisse reposer la mise en œuvre de l'adaptation telle que décrite par Lub.

4.2.2.1 Principes de l'extension pour des metalinks centrés objet

L'extension de Reflectivity à granularité objet repose sur la proposition de mise en œuvre énoncée à la section 4.1.1. La proposition consiste à migrer un objet de sa classe d'origine vers une sous-classe anonyme unique et dédiée à cet objet. Installer un metalink sur cette nouvelle classe revient alors à l'installer pour cet objet spécifique. La gestion de la migration de la création de la sous-classe de l'objet et de sa migration vers cette nouvelle classe deviennent parties intégrantes de la mécanique de Reflectivity − en toute indépendance de Lub et de sa mise en œuvre. Les principes de l'extension de Reflectivity pour des metalinks centrés objet ne sont

discutés dans la thèse que du point de vue de la mise en œuvre de Lub. Ces principes sont les suivants :

— Un gestionnaire de metalinks centrés objet, qui s'occupe de leur installation et de leur désinstallation. Ce gestionnaire maintient la compatibilité entre la base de Reflectivity et son extension. Notamment, toutes les possibilités des metalinks sont préservées. Par exemple, un metalink installé sur un objet a instance de A peut aussi être installé sur la classe B, c'est-à-dire que toutes les instances de B ainsi que l'instance a seront affectées par le metalink.
— Un gestionnaire de sous-classe anonyme, dont le rôle est de créer les sous-classes anonymes pour les objets sur lesquels placer un metalink. Ce gestionnaire est un sous-ensemble décorrélé de la notion d'adaptation (au sens Lub) du gestionnaire de sous-classe anonyme et de ses algorithmes décrits à la section 4.1.4.2.
— Un migrateur d'objet pour migrer les objets de leur classe d'origine vers la sous-classe anonyme unique dans laquelle est installé le metalink centré objet. Ce migrateur est un sous-ensemble décorrélé de la notion d'adaptation (au sens Lub) et de ses algorithmes décrits dans la section 4.1.4.3.

4.2.2.2 Mise en œuvre des metalinks centrés objets

Le gestionnaire de sous-classes anonymes Son rôle est la spécialisation de nœuds de l'AST pour des objets spécifiques. Suivant le principe de mise en œuvre décrit à la section 4.1.4.2, une sous-classe anonyme est créée pour chaque objet à spécialiser. Pour restreindre l'annotation d'un nœud à un objet, la méthode contenant ce nœud est dupliquée dans la sous-classe anonyme générée pour cet objet. Le nœud cible possède donc une copie parfaitement identique dans la nouvelle sous-classe anonyme de la classe d'origine de l'objet. L'algorithme de génération de sous-classes anonymes pour Reflectivity est illustrée par la Figure 4.12. La mise en œuvre de ce gestionnaire comprend également un registre d'optimisation pour la génération de sous-classes anonymes, tel que décrit par la Figure 4.7.

Migrateur d'objets Le migrateur d'objets pour Reflectivity est similaire dans le principe au migrateur décrit dans la section 4.1.4.3. Son rôle est de migrer les objets avec des metalinks dédiés vers leurs sous-classes anonymes.

Le gestionnaire de metalinks Son rôle est la gestion de l'installation et de la désinstallation des metalinks. Notamment, un metalink est toujours installé sur un nœud de l'arbre de syntaxe abstraite du programme, et non pas de l'objet ciblé. Les nœuds sont accessibles à partir du code source du programme, donc des classes. Le gestionnaire de metalinks, illustré par la Figure 4.12, se charge de retrouver la cible d'un metalink lors de son installation à partir d'un noeud de l'AST. Il assure

la liaison entre un nœud du programme et un nœud qui doit être annoté pour un objet spécifique – c'est-à-dire un nœud précédemment copié dans une sous-classe anonyme. Cette gestion comporte les responsabilités suivantes :

- Ordonner la création d'une sous-classe anonyme lors de l'installation d'un metalink, pour un triplet *{metalink, nœud, objet}*.
- Ordonner la migration d'un objet vers sa sous-classe anonyme, lors de l'installation d'un metalink.
- Retrouver un nœud spécifique dans une sous-classe anonyme, pour un couple *{nœud, objet}*. Si le nœud existe, alors la méthode contenant ce nœud a déjà été dupliquée dans la sous-classe, et il est directement possible d'installer un metalink sur ce nœud. Sinon, il faut copier la méthode contenant ce nœud dans la sous-classe (Figure 4.12).
- Résoudre les éventuels problèmes de lookup liés aux envois de messages à *super*, lors de l'installation d'un metalink (voir section 4.2.2.3).
- Ordonner la migration inverse de l'objet, de sa sous-classe anonyme vers sa classe d'origine. Cette migration inverse est effectuée lors de la désinstallation d'un metalink pour un triplet *{metalink, nœud, objet}*, si plus aucun metalink n'est installé sur cet objet.

Extension de l'API de Reflectivity Pour faciliter l'utilisation de l'extension, une interface utilisateur a été mise en œuvre. Cette interface est illustrée par la Figure 4.11. Elle fournit des méthodes pour l'installation de metalinks classiques (pour toutes les instances d'une classe) ainsi que pour les metalinks installés sur des instances particulières. L'API présentée se focalise sur les metalinks installés sur des nœuds de l'AST représentant des méthodes, cas qui nous intéresse particulièrement pour Lub.

```
1 metalink := [..." initialize metalink "...].
2 OrderedCollection link: metalink toMethodNamed: #add:.
3 set := Set new.
4 set link: metalink toMethodNamed: #add:.
5 metalink uninstallFrom: set.
6 metalink uninstallFrom: OrderedCollection.
7 metalink uninstall.
```

FIGURE 4.11 – API de l'extension de Reflectivity pour l'installation de metalinks sur des nœuds représentant des méthodes. L'API s'applique à une classe pour cibler toutes ses instances (ligne 2) ou à des objets particuliers (ligne 4).

On suppose qu'un metalink a été initialisé (ligne 1). L'interface décrite à la ligne 2 installe le metalink sur la méthode *#add*: de la classe *OrderedCollection*. Le nœud représentant cette méthode sera annoté pour toutes les instances de *OrderedCollection*, courantes et futures. La ligne 4 décrit l'interface pour l'installation du même metalink sur la méthode *#add*: d'un objet *set*, instance de *Set*. Seule cette instance sera affectée par ce metalink. Les lignes 5 et 6 désinstallent respectivement le metalink de l'objet *set* puis de la classe *OrderedCollection*. Alternativement, il est possible de désinstaller le metalink de toutes les entités pour lesquelles il était installé (ligne 7). Cette interface fait partie de l'implémentation de base de Reflectivity, et reste compatible avec son extension.

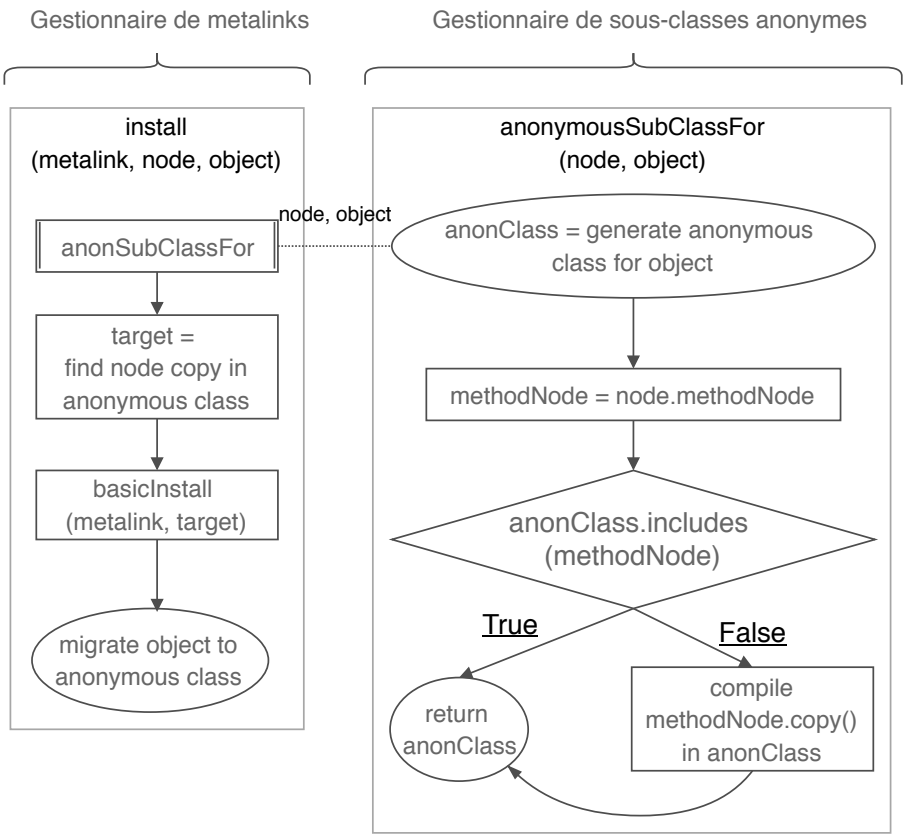


FIGURE 4.12 – Algorithmes de gestion de metalinks centrés objets suivant les principes généraux de Lub.

4.2.2.3 Problème du *super*

Le problème du *super* décrit à la section 4.1.3 nécessite une attention particulière. Il s'agit d'éviter qu'une méthode avec des envois de messages à *super*, sur laquelle un metalink dédié à un objet est installé, ne s'exécute plusieurs fois. Ce problème est résolu en mettant en place des sauts de lookup pour tous les envois de message à *super* dans une sous-classe anonyme. Le saut de lookup consiste à forcer le démarrage du lookup dans la super-super-classe de la classe anonyme (Figure 4.14). Tous les envois de message à *super* sont instrumentés par un metalink spécifique qui met en œuvre ce saut de lookup (Figure 4.13).

```
1  installSuperJumpLinksInMethodNode: node
2      (node allChildren select: [ :n | n isSuper ])
3          do: [ :superNode |
4              | messageSendNode superSuperClass link |
5              messageSendNode := superNode parent.
6              superSuperClass := node methodClass superclass
                   superclass.
7              link := MetaLink new.
8              link control: #instead.
9              link arguments: #(arguments receiver).
10             link selector: #value:value:.
11             link metaObject: [ :args :receiver |
12                     receiver perform: messageSendNode selector
13                              withArguments: args
14                              inSuperclass: superSuperClass ].
15             superJumpLinks add: link.
16             messageSendNode link: link ]
```

FIGURE 4.13 – Mise en œuvre du saut de *lookup* : tous les envois de messages à *super* (lignes 2 et 5) sont instrumentés avec un metalink (lignes 7 et 16). Ce metalink force le démarrage du *lookup* dans la super-super-classe de l'objet courant (lignes 12-14). Cette instrumentation est exclusivement mise en place dans les méthodes de sous-classes anonymes contenant des envois de messages au littéral *super*.

L'algorithme d'instrumentation contient deux étapes principales :

1. Trouver tous les envois de messages à *super* pour la méthode définie dans une sous-classe anonyme (ligne 2).
2. Création et installation d'un metalink (lignes 7-16) dont le méta-objet est un bloc (ligne 11). Le code de ce bloc (lignes 12-14) demande explicitement

à l'objet courant d'exécuter un sélecteur — celui du message originellement envoyé à *super* — avec ses arguments éventuels en démarrant le lookup dans sa super-super-classe — c'est-à-dire la classe originellement désignée par *super* avant la migration dans une sous-classe anonyme (Figure 4.14).

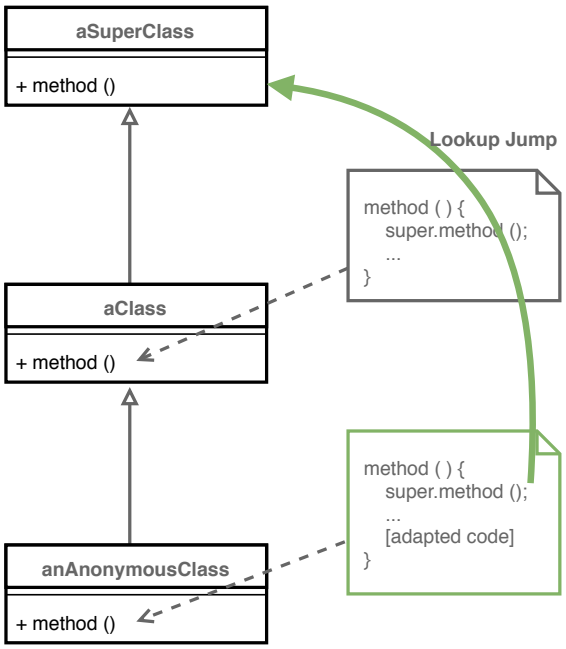


FIGURE 4.14 – Saut de lookup mis en œuvre avec Reflectivity pour résoudre le problème du *super*.

4.2.3 Bénéfices de l'extension de Reflectivity

Notre extension permettant d'installer des metalinks sur des objets spécifiques apporte les bénéfices décrits ci-dessous pour l'adaptation à granularité objet.

Unicité des objets ciblés. Il est déjà possible avec Reflectivity de cibler un objet précis avec un metalink, en utilisant une condition filtrant les objets de la classe où est installé le metalink. Toutes les instances de cette classe évaluent donc la condition pour savoir si le metalink les concerne. L'extension permet d'installer un metalink sur un objet spécifique, et n'affecte pas les autres instances de la même classe. Sémantiquement parlant, installer un metalink sur un objet signifie bien que seul cet objet sera affecté, et non pas tous les objets de la même classe avec une condition à évaluer pour filtrer l'objet.

Gain de performances. L'évaluation de l'impact sur les performances de Reflectivity est complexe. Il y a d'abord plusieurs niveaux mesurables. La mesure de l'impact sur un environnement de développement *live*, où un programme s'exécute en même temps qu'un-e développeur-euse l'instrumente avec des metalinks, n'est pas la même que la mesure de l'exécution brute d'un programme déjà instrumenté. Ensuite, les metalinks peuvent être paramétrés avec des options différentes ayant un impact sur leur exécution et qui peut faire varier les différences entre les metalinks centrés objet et les metalinks standard. Finalement, les résultats de la mesure sont difficiles à interpréter et une connaissance profonde de la mise en œuvre de Reflectivity et de son extension est nécessaire pour comprendre des éventuelles différences de temps d'exécution.

Un exemple de mesure a été effectué sur un ensemble d'un million d'instances de la même classe, dont une seule est affectée par un metalink. Ce metalink exécute un code simple, l'addition $2 + 3$, avant l'appel à une méthode m (contrôle $before$). Deux cas sont comparés :

— Dans le premier cas une condition est ajoutée au metalink et à chaque exécution de m la condition est évaluée pour savoir si le metalink concerne l'objet courant, en utilisant la version de base de Reflectivity.
— Dans le second cas, le metalink est uniquement installé sur l'objet ciblé en utilisant l'extension de Reflectivity.

Dans les mêmes conditions expérimentales (Intel Core i5-7300HQ 4x2.50GHz, Debian GNU/Linux 9 - stretch 64, Pharo-7.0 64 Bit), l'extension avec le metalink centré objet est jusqu'à 20% plus rapide que le metalink avec une condition pour déterminer l'objet cible. La modification des options du metalink peut augmenter ou diminuer ce gain de performances. Une étude approfondie de l'extension sur les performances de Reflectivity permettrait donc de préciser les différents impacts selon les cas d'utilisation et le paramétrage des metalinks.

Mécanisme d'adaptation par objet intégré au langage. Reflectivity et son extension forment un mécanisme d'adaptation à granularité objet, sur lequel peut reposer la mise en œuvre d'un patron de langage pour l'adaptation non-anticipée de comportement tel que Lub.

4.2.4 Modèle de mise en œuvre de Lub basé sur Reflectivity

Le modèle de mise en œuvre de Lub se base sur une utilisation du gestionnaire de metalinks au travers de l'API étendue de Reflectivity. Lub étend le gestionnaire de sous-classe anonyme et le migrateur d'objet de Reflectivity au concept d'adaptation. Du point de vue de Lub, Reflectivity est un mécanisme d'adaptation de plus bas niveau permettant de mettre en œuvre l'adaptation non-anticipée telle que dé-

finie dans le patron. Il s'agit d'une mise en œuvre possible suivant les principes et techniques décrits depuis le début du chapitre.

L'adaptation. Une méta-adaptation est modélisée par une classe du langage, et son instanciation produit des adaptations uniques. L'adaptation est elle-même instanciée lorsqu'elle est appliquée à un objet. Pour chaque objet adapté, une instance différente de l'adaptation appliquée est générée. Une API de définition d'adaptation à partir de la méta-adaptation est implémentée, mais sa syntaxe dépend du langage hôte. Les API implémentées dans les langages Python et Pharo sont décrites dans la section 4.2.5 à suivre.

Le gestionnaire de sous-classe anonymes de Reflectivity étendu à l'adaptation. Les adaptations instanciées sont compilées dans les sous-classe anonymes correspondantes fournies par le gestionnaire de Reflectivity. Il s'agit d'une extension immédiate de l'algorithme présenté par la Figure 4.12 pour l'installation de metalinks centrés objet par celui de la Figure 4.7 pour la compilation d'adaptation.

L'adaptateur d'objet. L'adaptateur, tel que décrit dans la section 4.1.4.1, pilote l'adaptation des objets. Il dispose d'une API de contrôle pour appliquer les adaptations aux objets. Ici, il s'agit d'une extension légère du gestionnaire de metalinks de Reflectivity. L'adaptation d'un objet se passe en deux étapes :

1. L'adaptateur demande une sous-classe anonyme pour l'objet ciblé par l'adaptation. Les méthodes apportées ou modifiées par l'adaptation sont compilées dans cette sous-classe, (comme décrit à la section 4.7). Ces méthodes sont compilées sous un alias : une méthode m sera compilée dans la sous-classe sous le nom $adapt_m$.
2. L'adaptateur ordonne l'installation d'un metalink centré objet sur la méthode m. Ce metalink spécifie le remplacement de l'exécution de la méthode m de l'objet adapté par l'exécution de la méthode $adapt_m$, contenant le comportement adapté. Deux cas sont pris en compte :
   — Si cette méthode m existe déjà dans la classe d'origine de l'objet adapté, alors l'algorithme appliqué est celui de la copie de nœuds de l'extension de Reflectivity.
   — Si la méthode m n'existe pas, alors une méthode m vide est compilée dans la sous-classe anonyme pour permettre l'installation du metalink.

   Les opérations de migration d'objets vers leur sous-classe anonyme ou leur classe d'origine sont entièrement prises en charge par l'extension de Reflectivity.

Le metalink mettant concrètement en œuvre une adaptation est décrit par la Figure 4.15. On suppose qu'un objet nommé *object* doit être adapté. Son adaptation

spécifie que la méthode m doit être adaptée par la méthode $adapt_m$. Le metalink spécifie un contrôle de type *instead* (ligne 1). L'exécution du nœud sur lequel le metalink sera installé sera remplacée par l'envoi du sélecteur $adapt_m$ (ligne 3) au méta-objet représenté par l'objet adapté (ligne 2). Le metalink est installé sur le nœud de la méthode m pour l'objet ciblé (ligne 5). Un tel metalink est instancié et configuré pour chaque méthode spécifiée pour l'adaptation d'un objet.

```
1 metalink control: #instead.
2 metalink metaObject: object.
3 metalink selector: #adapt_m.
4 metalink arguments: #(arguments).
5 object link: metalink toMethodNamed: #m.
```

FIGURE 4.15 – Configuration d'un metalink pour l'adaptation d'une méthode m par une méthode $adapt_m$ sur l'objet *object*. Les sélecteurs $\#m$ et $\#adapt_m$ sont fournis par l'adaptation, qui spécifie comment l'objet est adapté.

4.2.5 Interfaces pour l'adaptation

Une API pour Lub a été développée pour sa mise en œuvre dans le langage Pharo [1]. Il s'agit d'un ensemble d'interfaces très restreint, mais qui permet d'exploiter toutes les fonctionnalités apportées par Lub tout en restant strictement dans le périmètre du paradigme objet. Au delà des concepts dont le développeur ou la développeuse a besoin de manipuler pour adapter des objets, l'API fournit des éléments pour faciliter l'utilisation de Lub en pratique − et notamment des facilités syntaxiques. Les interfaces pour définir des adaptations et pour les appliquer à des objets sont décrites dans les paragraphes suivants.

API pour la définition d'adaptation. L'API de définition d'adaptation est décrite par la Figure 4.16, et contient deux méthodes : une méthode de création d'adaptation (ligne 2) et une méthode d'accès à une adaptation (ligne 4). La création d'adaptation se fait en passant les paramètres suivants : un nom ($aName$), un tableau contenant les spécifications des méthodes adaptées ($structuralFeaturesArray$) et la classe où trouver ces méthodes ($aClass$). Le tableau $structuralFeaturesArray$ spécifie comment sont adaptées les méthodes à partir de celles trouvées dans $aClass$. Ce tableau contient des associations constituées d'un nom de méthode et d'un contrôle. Un contrôle spécifie comment une méthode de

1. https://github.com/StevenCostiou/Lub

aClass modifiera la méthode de même nom de l'objet qui sera adapté. Les différents types de contrôle sont hérités de Reflectivity, et peuvent être des types suivants :

1. *before* : La méthode ciblée de *aClass* s'exécutera avant la méthode portant le même nom de l'objet adapté.
2. *instead* : La méthode ciblée de *aClass* s'exécutera à la place de la méthode portant le même nom de l'objet adapté.
3. *after* : La méthode ciblée de *aClass* s'exécutera après la méthode portant le même nom de l'objet adapté.
4. *acquire* : Nouveau contrôle ajouté par Lub, et permet à un objet d'obtenir une nouvelle méthode provenant de la classe *aClass* ciblée par l'adaptation.

```
1 Adaptation class>>
2     named: aName adapt: structuralFeaturesArray with: aClass
3 Adaptation class>>
4     named: aName
```

FIGURE 4.16 – Lub, API de définition d'adaptations.

La Figure 4.17 illustre l'utilisation des contrôles dans la spécification et la définition d'une adaptation.

```
1 Adaptation named: 'ExampleAdaptation'
2           adapt: {#printString->#before. #crLog:->#instead.}
3           with: Transcript
```

FIGURE 4.17 – Lub, illustration d'une adaptation. Les méthodes *printString* et *crLog* de l'objet qui sera adapté seront respectivement précédées par l'exécution de la méthode *printString* de la classe *Transcript* et remplacée par l'exécution de la méthode *crLog* de la classe *Transcript*.

API pour l'application d'adaptation. Les méthodes de l'interface d'adaptation d'objets sont décrites par la Figure 4.18. Ces méthodes sont accessibles à tous les objets du système, et n'importe quel objet peut donc être soumis à l'adaptation. Dans le code présenté en illustration, l'interface est appliquée à un objet anonyme représentant n'importe quel objet possible et nommé *object*. Le message *adaptWith*: (ligne 1) prend en paramètre une adaptation (ici retrouvée par son nom unique),

et a pour effet d'adapter l'objet recevant le message. Le message *isAdapted* (ligne 2) permet de connaître le statut d'un objet : *true* si ce dernier est actuellement adapté et $false$ sinon. Finalement, il existe trois possibilités pour restaurer un objet. L'interface *adaptWith*: (ligne 1) peut être utilisée avec en paramètre le littéral *nil*. Toute adaptation est retirée de l'objet. L'interface *unadapt* est un raccourci syntaxique permettant la même opération (ligne 4). Enfin, l'interface *revert*: prend en paramètre une adaptation précise, et provoque la restauration d'un objet modifié par cette adaptation (ligne 3). Cette interface permet de rendre explicite la sémantique de la restauration d'objet à la lecture du code, et de savoir quelle adaptation est retirée de l'objet ciblé.

```
1 object adaptWith: (Adaptation named: aName).
2 object isAdapted. ">>> true"
3 object revert: (Adaptation named: aName).
4 object unadapt.
```

FIGURE 4.18 – Lub, API d'adaptation d'objets.

4.3 Application du patron de langage : une mise en œuvre en Python avec les *Talents*

Lub étant un patron de langage, il doit pouvoir être appliqué à d'autres langages satisfaisant aux contraintes n^{o}1 et n^{o}2 énoncées dans ce chapitre (section 4.1.1). C'est le cas de Python [Python, 2017], qui possède des capacités réflexives proches de celles de Pharo. Suivant les principes de mise en œuvre proposés dans la section 4.1, nous avons pu étendre le langage Python avec Lub [2].

Nous utilisons les *Talents* [Ressia et al., 2014] pour le mécanisme d'adaptation d'objets sous-jacent. Un Talent permet à un objet d'acquérir ou de perdre du comportement, ainsi que de composer des unités de comportement dans une seule adaptation comportementale. Un Talent peut aussi permettre à un objet d'acquérir de nouveaux états. Nous utilisons une implémentation *ad-hoc* inspirée des Talents pour mettre en œuvre un adaptateur d'objets, un gestionnaire de sous-classes anonymes ainsi qu'un migrateur d'objets tels que décrits dans les principes de mise en œuvre du patron de langage. À l'instar des Talents, cette version de Lub permet à un objet d'acquérir de nouveaux états, tel que cela est décrit dans sa structure de langage et dans sa sémantique opérationnelle informelle décrite dans le chapitre 3.

2. https://git.kloum.io/StevenCostiou/PyLub

Cette mise en œuvre est cependant incomplète. Il s'agit d'une expérimentation destinée à explorer des implémentations de Lub basées sur d'autres supports que Reflectivity. En particulier, nous ne proposons pas de solution au problème du *super*, ni de gestion des adaptations gardées. Il n'y a pas non plus la même souplesse qu'avec l'utilisation de Reflectivity comme mécanisme d'adaptation. Il n'est pas possible, par exemple, de simplement spécifier un préfixe ou un suffixe à une méthode ($before$, $after$). Pour rajouter de tels préfixes et suffixes dans une méthode adaptée, le code originel de la méthode d'origine doit être intégralement copié dans la méthode adaptée. Ces limitations sont néanmoins du domaine de la mise en œuvre, et des solutions pourraient être développées pour satisfaire une implémentation plus complète de Lub dans le langage Python. L'API d'utilisation de Lub pour l'adaptation d'objets est en revanche similaire à celle implémentée dans Pharo, au détail de syntaxe du langage près.

4.4 Analyse des performances

Dans cette section, une évaluation des performances de Lub est effectuée pour le langage Pharo. Les résultats sont comparés avec les performances des mécanismes natifs de Pharo. Les éléments suivants sont comparés : la vitesse du lookup, le temps d'adaptation des objets et la consommation mémoire. Les tests effectués sont reproductibles [3], et les résultats présentés sont publiquement disponibles [4]. Dans cette section, nous utiliserons le terme *Lub* pour désigner notre mise en œuvre Pharo de Lub.

4.4.1 Critères de comparaison

L'analyse des performances est basée sur deux versions d'un code de référence contenant 5 méthodes. La première est un code étendu avec du comportement adapté de manière anticipée, sous forme de sous-classes réimplémentant les méthodes adaptées. Les objets sont donc directement instanciés à partir des classes adaptées déjà présentes dans le système. Cette version représente donc un programme adapté par les moyens conventionnels du paradigme objet, c'est-à-dire l'héritage. La seconde version est composée du code de référence et des adaptations définies avec Lub pour altérer le comportement des objets, de manière identique aux modifications appliquées par les sous-classes insérées dans la première version. Cette seconde version représente un programme adapté de manière non-anticipée. L'objectif est d'évaluer les différences de performances entre un programme adapté dynamiquement par Lub et un programme adapté statiquement avec les moyens natifs Pharo.

3. https://github.com/StevenCostiou/Lub
4. https://github.com/StevenCostiou/Lub/tree/master/benchmarks-results

4.4.2 Protocole de la prise de mesure

Toutes les mesures sont prises sur une machine Intel Core i5-7300HQ 4x2.50GHz, Debian GNU/Linux 9 - stretch 64, 8GB RAM et sous Pharo-7.0 [5] 64 Bit.

Les scripts des mesures de vitesse (migration et lookup) sont illustrés ci-dessous. Les mesures sont prises par l'envoi du message *timeToRunWithoutGC* à un bloc contenant le code dont on veut mesurer la vitesse d'exécution. Le temps mesuré exclut le temps utilisé par le gestionnaire de mémoire de Pharo. Cette prise de mesures est démarrée dans un processus avec priorité maximale (*Processor timingPriority*). Les mesures ne comprennent pas le temps d'instantiation des objets, qui est effectuée avant le démarrage du script. Un script est dédié à chaque type de mesure (Pharo, Lub) et à chaque aspect mesuré (temps de migration, consommation mémoire, temps d'exécution du lookup). Un nouvel environnement Pharo est ouvert pour chaque script, et terminé après que les résultats sont sauvegardés dans des fichiers [6].

```
1 ...objects instantiation...
2 [[...benchmark code...] timeToRunWithoutGC]
3     forkAt: Processor timingPriority
```

4.4.3 Périmètre des mesures

Certains éléments, comme des étapes du processus d'adaptation ou certaines parties du code exécutable, ne sont pas mesurés dans ces évaluations. Et notamment :

- Le temps nécessaire à la conception et à l'injection d'adaptation dans le programme en cours d'exécution n'est pas mesuré ni évalué. Lub ne propose pas ces fonctionnalités en tant que tel et se focalise uniquement sur l'extension du langage à l'adaptation non-anticipée de comportement. Le choix des outils ou des techniques mises en œuvre pour interagir avec le programme, comme par exemple les dévermineurs à distance [Papoulias et al., 2015, TelePharo, 2018], a un impact sur la performance générale du programme. C'est un aspect complètement indépendant des ralentissements éventuellement produits par le mécanisme d'adaptation en lui-même.
- Le coût du parcours des structures de données contenant les entités à adapter n'est pas mesuré. Il s'agit typiquement du temps passé dans les boucles pour appeler les méthodes pour lesquelles les différences de vitesse dues au lookup instrumenté sont mesurées. Dans tous les cas, un temps similaire est passé dans les boucles par Pharo et par Lub pour itérer sur ces structures de données.

5. Pharo-7.0+alpha.build.1095.sha.9fd3a4b2a61974f4de95434be3ee172901541af7
6. https://github.com/StevenCostiou/Lub/tree/master/benchmarks-results

4.4.4 Évaluation de la vitesse du lookup instrumenté

Conceptuellement, l'adaptation par Lub est effectuée par une indirection du lookup. Le temps que prend le lookup pour retrouver la méthode recherchée est donc important pour saisir l'impact de l'adaptation sur la vitesse d'exécution du programme. Pour mesurer cet impact, nous mesurons la différence du temps d'exécution entre deux méthodes implémentant le même code. L'une de ces méthodes est statiquement définie dans la classe de l'objet adapté, l'autre méthode est adaptée par Lub. Est posée l'hypothèse que le même code pour deux méthodes différentes aura un temps d'exécution similaire. La différence en terme de temps d'exécution de ces deux méthodes correspondra, dans nos scénarios, au temps supplémentaire induit par l'instrumentation du lookup.

Les critères d'évaluation de la vitesse du lookup sont définis dans la Table 4.1. Le temps mesuré est le temps d'exécution de la même méthode un million de fois. Cette mesure est effectuée 1000 fois pour le lookup Pharo et le lookup Lub. Les données présentées sont, pour chaque couple de mesures, une moyenne avec leur intervalle de confiance (95%). La différence entre les deux lookups est exprimée en comparant l'intervalle de confiance pour le lookup Lub à la moyenne du lookup Pharo, pour chaque scénario.

Méthode	**Critère : appel de méthode**	**Code Source**
m	Appel direct d'une méthode non adaptée	100 printString
m1	Appel direct d'une méthode adaptée	100 printString
m21	Appel d'une méthode adaptée à partir d'une méthode non adaptée	self m1
m22	Appel d'une méthode adaptée à partir d'une méthode adaptée	self m1
m4	Appel direct d'une nouvelle méthode	100 printString

TABLE 4.1 – Scénarios d'évaluation du lookup

Les résultats sont présentés dans la Table 4.2. Globalement, les différences entre les temps d'exécution des deux lookups sont toujours inférieures à 12%. Lub est légèrement plus rapide ou aussi rapide pour les méthodes non adaptées ou les méthodes acquises (m et $m4$). Ce sont des méthodes qui ne nécessitent pas d'indirection de lookup : soit la méthode existe déjà, soit elle est ajoutée. Pour les méthodes faisant appel à une méthode adaptée ($m1$, $m21$ et $m22$), le temps d'exécution du lookup avec Lub est de 10.70% à 12.04% plus lent. La plus grande différence – de 11.49% à 12.04% de temps d'exécution supplémentaire – concerne les méthodes adaptées qui font appel à une méthode adaptée ($m22$). Le temps d'exécution de Lub est sen-

siblement plus lent lorsqu'une ou plusieurs indirections de lookup sont effectuées à l'appel d'une méthode.

lookup	**Pharo** (en ms)	**Lub** (en ms)	**Différence** [a]
m	93.58± 0.76	88.6± 0.25	-5.58%−-5.05%
m1	90.79± 0.32	101.05± 0.24	11.04%−11.57%
m21	93.09± 0.35	103.25± 0.21	10.70%−11.14%
m22	93.11± 0.27	104.06± 0.26	11.49%−12.04%
m4	90.31± 0.2	91.14± 0.32	0.57%−1.28%

[a] Intervalle de confiance : 95%

TABLE 4.2 – Comparaison des vitesses d'exécution du lookup pour 10^6 exécutions (en millisecondes)

Une difficulté lors de l'interprétation de ces mesures est, selon l'hypothèse que nous avons formulée, que le temps d'exécution d'un même code est supposé constant entre différentes exécutions. Cela n'est pas vrai en pratique pour le lookup de Pharo. Des mesures comparées de ce lookup, suivant les mêmes scénarios, ont montré que des différences de temps d'exécution pouvaient exister nativement. Par exemple, le lookup de Pharo peut être en moyenne jusqu'à 5% plus lent entre deux ensembles de 1000 mesures différentes. De fait, certains scénarios ont parfois donné des différences de performances variables entre le lookup Lub et le lookup Pharo. Le fait que le lookup Lub soit plus rapide que le lookup Pharo (cas de la méthode non adaptée m) n'est donc pas notable en soi. En revanche, les mesures pour les méthodes adaptées ou faisant appel à une méthode adaptée ($m1$, $m21$, $m23$) montrent systématiquement un ralentissement de 10 à 12% de la vitesse d'exécution du lookup. Les résultats présentés dans la Table 4.2 représentent un cas typique de perte de performances lors de l'utilisation de Lub, où les différences les plus notables portent toujours sur les scénarios impliquant des appels de méthodes adaptées.

4.4.5 Évaluation de la vitesse d'adaptation des objets

Lorsqu'un objet est adapté, il est migré vers une sous-classe anonyme de sa classe d'origine : il peut alors acquérir ou modifier du comportement. Effectuer cette opération de migration a un coût, et peut ralentir le système en cours d'exécution.

Une première évaluation étudie le coût de la migration de 1000 objets suite à leur modification par une même adaptation. Une seconde évaluation évalue le coût de migration de 1000 objets modifiés par une adaptation unique. Dans ce second cas les 1000 objets sont adaptés chacun par une adaptation comportementale différente. Les résultats sont visibles dans la Table 4.3. Les mêmes évaluations sont effectuées

à la suite pour étudier l'impact de restaurer les objets, c'est-à-dire d'annuler les adaptations. Ces résultats sont visibles dans la Table 4.4.

Tous les objets sont des instances d'une classe C, et possèdent deux méthodes $m1$ et $m2$. Les adaptations ajoutent deux nouvelles méthodes $m3$ et $m4$ et modifient le comportement des méthodes $m1$ et $m2$. Les éléments suivants sont mesurés :

— Le temps d'exécution nécessaire pour migrer les objets vers leurs versions adaptées.

— Le temps d'exécution passé dans la mécanique interne de Lub : compilation de code, installation de metalinks, création de sous-classes anonymes, etc.

1000 objets	**Migration**	**Compilation**	**Total**
1 adaptation$^\bullet$	$\leq$2	$\approx$50	$\approx$52
1000 adaptations*	$\approx$14	9932	9946

$^\bullet$ 1000 objets adaptés par la même adaptation

* 1000 objets adaptés chacun par une adaptation différente

Table 4.3 – Temps d'exécution de l'adaptation d'objets (en millisecondes)

Le temps nécessaire pour la migration d'objets dépend de l'homogénéité des adaptations qui leurs sont appliquées. Le coût de migration est considérablement plus faible lorsque tous les objets sont adaptés par la même adaptation, que lorsque chaque objet est adapté par une adaptation spécifique (Table 4.3). Abstraction faite des temps de compilation, les temps de migration des objets ne sont pas significativement différents pour 1 ou 1000 adaptations. Le coût de migration en lui-même est négligeable par rapport au temps passé dans la compilation et la gestion des metalinks. Le goulot d'étranglement se situe au niveau du temps de compilation qui dépend directement du nombre d'adaptations différentes qui sont appliquées. Les opérations d'optimisation du gestionnaire de sous-classe anonymes (Figures 4.7 et 4.12) facilitent l'adaptation d'ensembles d'objets homogènes (même classe, même adaptation). Le coût de compilation à l'exécution est alors considérablement réduit. Cet avantage disparaît dès que les ensembles d'objets ou d'adaptations à appliquer deviennent plus hétérogènes. Concrètement, un grand nombre d'objets de classes différentes ou d'adaptations différentes augmentent le temps de compilation.

Des optimisations supplémentaires pourraient être mises en place dans le gestionnaire de metalinks pour accélérer les temps de compilation. Les opérations de compilation des classes anonymes et de gestion des metalinks centrés objet pourraient être effectuées *hors-ligne*, avant d'injecter les sous-classes et les metalinks dans le programme. Le coût de ces opérations pourrait alors être réduits à zéro pour le programme en cours d'exécution. La seule opération restant à charge du système serait la migration des objets vers leur nouvelle classe. Cela reposerait

nécessairement sur un outillage, qui aurait également un coût d'utilisation. Le choix d'une telle solution dépendrait du gain obtenu par rapport à une compilation des sous-classes anonymes et à une gestion des metalinks au sein même du programme en cours d'exécution.

1000 objets	**Migration**	**Compilation**	**Total**
1 adaptation•	≤2	0	0
1000 adaptations⋆	≤2	0	0

• 1000 objets adaptés par la même adaptation
⋆ 1000 objets adaptés chacun par une adaptation différente

TABLE 4.4 – Temps d'exécution de la restauration d'objets à leur comportement initial (en millisecondes)

Le coût de la restauration d'objet à leur comportement d'origine (Table 4.4) est très faible comparé au coût de la migration d'objets. Aucune compilation n'est nécessaire, et le temps mesuré est le temps brut utilisé pour les primitives du système et la migration d'objets vers leur classe d'origine.

Ces tests et évaluations de performances pour la migration et la restauration d'objets sont reproductibles à partir du code du *package benchmarks* disponible publiquement en ligne [7].

4.4.6 Évaluation de la consommation mémoire des objets adaptés

Pour évaluer la consommation mémoire supplémentaire due à l'adaptation non-anticipée d'objets, deux programmes sont comparés. Dans un premier programme, l'adaptation est mise en œuvre avec Lub. Dans le second, les objets sont adaptés par le biais d'une mise à jour du programme. Cette mise à jour produit, dans les cas les plus extrêmes, une version spécialisée de chaque instance adaptée – c'est-à-dire une classe dédiée par objet. L'objectif est l'étude de la surconsommation de la mémoire induite par Lub en comparaison de cette version hypothétique de l'application, adaptée par les moyens natifs du langage. La consommation mémoire est mesurée une fois que tous les objets surveillés ont été adaptés.

Deux scénarios sont présentés, dans lesquels une classe originelle $C0$ possède deux méthodes $m1$ et $m2$. Ces deux méthodes exécutent l'instruction *100 printString*. 1000 instances de cette classe existent à l'exécution, et doivent être adaptées.

7. https://github.com/StevenCostiou/Lub/tree/master/LUB-Benchmarks.package

Un moyen simple serait de spécialiser la classe $C0$, en définissant autant de sous-classes que d'adaptations différentes. Pour des raisons de simplicité et faciliter la comparaison avec Lub, toutes les adaptations et sous-classes implémentent le même comportement que la méthode $m1$ de $C0$. L'adaptation ajoute deux nouvelles méthodes, $m3$ et $m4$ et modifient le comportement de $m1$ et $m2$.

Les mesures sont effectuées avec l'outil *SpaceTally* de Pharo [Bergel et al., 2013]. Sont mesurés l'espace pris en mémoire par les classes additionnelles, notamment le code des méthodes compilées, ainsi que la taille en mémoire prise par les instances adaptées de ces classes.

Scenario 1$^\bullet$ Tous les objets doivent être adaptés avec les mêmes variations comportementales. $C1$ est une sous-classe de $C0$, qui implémente ces comportements adaptés. Une fois que tous les objets sont migrés vers leur nouvelle classe $C1$, l'espace qu'ils occupent en mémoire est mesuré. Puis dans un second temps, les mêmes variations de comportement sont mise en œuvre avec Lub, par le biais d'adaptations appliquées au même nombre d'instances de $C0$. La mesure est à nouveau répétée.

Scenario 2* Pour fournir une adaptation de comportement spécifique à chaque objet via les mécanismes de l'orienté objet, une classe spécifique est créée pour chaque objet. Il existe donc 1000 classes, chacune dédiée à l'adaptation d'un objet. Chaque objet est migré vers sa nouvelle classe. De manière similaire avec Lub, 1000 adaptations différentes adaptent chacune un objet précis. Les mesures sont répétées.

Les résultats de l'évaluation mémoire sont présentés dans la Table 4.5. Dans le cas du premier scénario, Lub introduit une différence en mémoire de +148% par rapport à une version adaptée par Pharo. Lorsque le nombre d'instances adaptées augmente, cette utilisation mémoire supplémentaire reste constante (651 octets). Le surcoût relatif, en revanche, tend à diminuer jusqu'à devenir négligeable (+35% pour 100 objets puis +4% pour 1000 objets). Les optimisations du gestionnaire de sous-classes anonymes permettent de payer le coût mémoire lorsque la première sous-classe anonyme est générée pour la même adaptation. Le coût restant à chaque adaptation du même type d'objet est l'espace mémoire occupé par les instances elles-mêmes, qui est équivalent à celui occupé par des objets avec la même structure de classe et le même état.

Cependant, dans le cas du second scénario où les objets sont tous adaptés par une adaptation particulière (ensembles hétérogènes d'adaptations), le surcoût relatif en mémoire supplémentaire est supérieur (+220%) mais reste constant quelque soit le nombre d'objets adaptés. Le surcoût mémoire absolu, lui, augmente avec le nombre d'instances adaptées. Ce coût est dû au nombre de méta-entités (classes anonymes, méthodes, etc.) qui sont créées ou injectées pour produire le

Instances	**10•**	**100•**	**1000•**	**10⋆**	**100⋆**	**1000⋆**
Pharo	440	1880	16280	2960	29600	296000
Lub	1091	2531	16931	9470	94700	947000
Différence	+148%	+35%	+4%	+220%	+220%	+220%

• La même adaptation pour tous les objets
⋆ Une adaptation spécifique pour chaque objet

TABLE 4.5 – Comparaison de la consommation mémoire de 1000 objets adaptés par Lub avec 1000 objets adaptés par héritage dans Pharo (en octets).

comportement correspondant à une adaptation spécifique. Pour une adaptation donnée, ajouter ou adapter une méthode représente un coût fixe en mémoire, dépendant de sa spécification. Ainsi, produire de nombreuses adaptations différentes ajoutera autant de surplus mémoire au système. Adapter un grand nombre d'objets de structure différente, c'est-à-dire des instances de différentes classes, aura également un impact significatif sur la consommation mémoire du programme en cours d'exécution.

4.4.7 Limites de l'évaluation des performances et de la consommation mémoire

Il est important de noter que ces mesures ne sont valides qu'à titre illustratif, et pour les scénarios présentés qui ne représentent que des cas d'adaptation simples et limitées de comportement. Le principe de Lub étant d'adapter des objets de manière complètement non-anticipée, il n'est pas possible de prédire de manière générale quel serait l'impact d'une adaptation sur un programme en cours d'exécution. Les éléments suivants peuvent produire des variations non négligeables dans les mesures présentées :

Sur la vitesse d'exécution d'une méthode adaptée. Il est entendu que la vitesse d'exécution d'une méthode adaptée dépend du code exécuté. Si une adaptation produit une modification importante par de l'ajout de comportement, ou par une réduction de comportement, le code adapté variera en fonction de cette modification et pourra être respectivement plus lent ou plus rapide. Les mesures présentées dans la section 4.4.4 font abstraction de cette notion, et montrent spécifiquement la différence de temps d'exécution pour le lookup.

Sur l'impact en mémoire d'un ensemble d'objets adaptés. De la même manière, la variabilité du code adapté peut modifier l'impact de l'adaptation sur la mémoire. Par exemple, de nombreuses méthodes étendues par des modifications

de comportement lourdes et complexes peuvent augmenter les différences mesurées lors de nos évaluations. En revanche, et une fois que les adaptations sont en place, le coût mémoire supplémentaire dépend de l'hétérogénéité des ensembles d'objets et de leurs adaptations.

Pour un cas d'utilisation précis et pour un ensemble fixe d'adaptations et d'objets connus, il serait possible de calculer l'impact de l'application des adaptations sur les objets du programme en cours d'exécution. Une telle analyse nécessiterait des modèles et des outils dédiés, et devrait être effectuée avant chaque adaptation du programme. Cela nécessite également l'apparition du besoin d'adaptation, et ce dernier étant imprévisible, cette d'analyse ne pourrait être qu'effectuée de manière *ad-hoc*.

Sur les optimisations de la mise en œuvre. Dans la mise en œuvre présentée, le choix est fait d'optimiser les temps de migration pour les ensembles homogènes d'objets et d'adaptations. Cette optimisation consiste à économiser les temps de compilation dans le gestionnaire de sous-classes anonymes, et les résultats de l'optimisation sont visibles dans la Table 4.3. Optimiser de telle manière le système est un choix contestable. Des expérimentations d'optimisation des temps d'exécution du lookup ont pu réduire le temps supplémentaire induit par l'adaptation. Le ralentissement peut alors être limité à un maximum de 4% au lieu de 12% dans la mise en œuvre présentée (Table 4.2). Mais ces optimisations sont incompatibles avec celles du gestionnaire de sous-classes anonymes, et rallongent fortement les temps de compilation − de 50ms à 10 secondes dans la Table 4.3. Il s'agit d'un équilibre à trouver entre le bénéfice recherché (par exemple un lookup performant) et les effets négatifs apportés par une optimisation (par exemple des temps de compilation longs). Idéalement, en se projetant dans la perspective de la réalisation d'un outil industriel, ce sont des optimisations qui pourraient être paramétrables selon le cas d'utilisation et les contraintes du contexte (environnement de développement/environnement de production). L'obtention d'un lookup performant dépend de l'outillage et de sa capacité à décorréler la compilation de l'environnement d'exécution du programme.

4.5 Adaptation non-anticipée de comportement dans une flotte de drones

Cette section présente un scénario d'adaptation dans une simulation d'une flotte de drones. L'objectif n'est pas de démontrer l'utilisabilité de Lub dans un cas réel d'adaptation de drone, mais d'illustrer comment code et comportement pourraient être adaptés à l'exécution pour du déverminage, d'un point de vue purement logiciel.

C'est donc une simulation conçue dans le cadre de cette thèse, pour mettre l'accent sur les possibilités d'investiguer et de modifier un système en cours d'exécution au travers d'adaptations non-anticipées de comportement. Cette simulation et les adaptations de comportements pour en corriger les problèmes sont mis en œuvre avec Pharo.

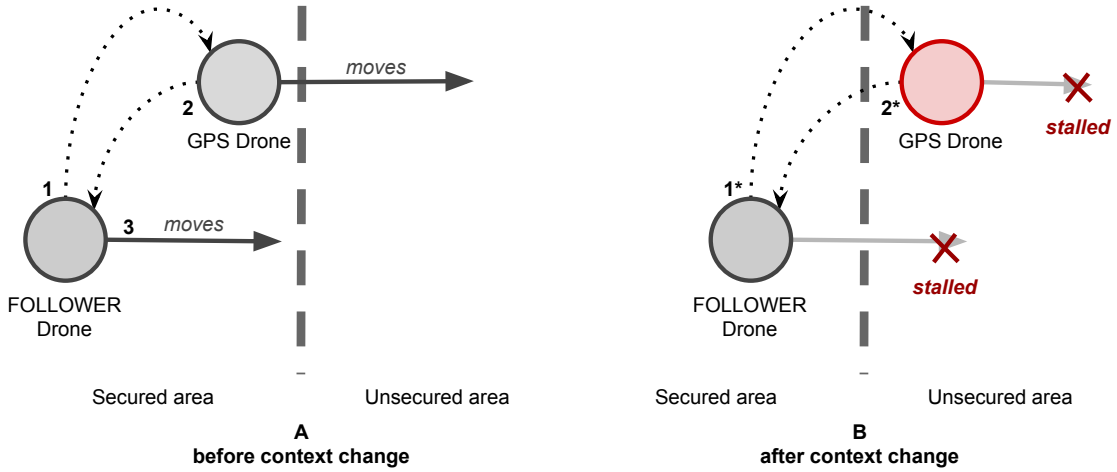


FIGURE 4.19 – La flotte se déplace dans une zone sécurisée (fig. A). *Follower Drone* demande la position GPS (1) de *GPS Drone* qui lui répond (2) et la flotte peut se déplacer (3). Quand *GPS Drone* entre dans la zone non sécurisée (fig. B), ce dernier commence à chiffrer ses communications. Quand *Follower Drone* envoie une requête (1*), *GPS Drone* répond alors avec des données chiffrées (2*). *Follower Drone* ne s'est pas rendu compte du changement de contexte, et ne peut pas interpréter les données chiffrées. La flotte est bloquée.

4.5.1 Scénario : la perte de GPS

Dans ce scénario, deux drones se déplacent ensemble, proches l'un de l'autre. Le premier, nommé *gps-drone*, possède un GPS actif tandis que le second, nommé *follower-drone*, utilise un canal de communication avec *gps-drone* pour récupérer une position relative. Le partage des données GPS entre les deux drones permet à la flotte de se déplacer. Comme l'environnement est complètement dynamique, ce dernier présente des incertitudes : il n'est pas possible de prévoir tout ce qui pourrait arriver pendant le vol, et donc de prévoir tous les cas possibles de comportements. Notre scénario est illustré par la Figure 4.19. La flotte entre dans une zone non sécurisée où toutes les communications doivent être chiffrées. Les deux drones entrent cependant dans un mode d'attente car *follower-drone* échoue à calculer sa position à partir des données GPS de *gps-drone*. Le signal GPS de *gps-drone* est pourtant

stable et la communication entre les deux drones est fonctionnelle. Le changement de contexte est effectif pour *gps-drone*, mais pas pour *follower-drone* qui ne sait pas que les données reçues doivent être déchiffrées. Les drones sont donc en attente d'un déblocage de la situation : c'est un comportement prédéfini. Dans cette simulation il s'agit d'un *bug*. C'est donc par définition une situation non-anticipée qui n'a pas été prévue lors de la conception du système.

4.5.2 Stratégie de déverminage : l'adaptation non-anticipée de comportement

La flotte est bloquée à cause d'un des drones qui n'a pas adapté son comportement au changement de contexte (l'entrée dans la zone non sécurisée). Il s'agit d'un problème imprévu dont la raison ne semble pas évidente au premier abord. Les options sont également limitées pour un opérateur ou une opératrice. Soit les drones sont rappelés pour une analyse hors ligne du logiciel et des données accumulées pendant le vol, soit le problème doit être corrigé dynamiquement. Le logiciel en lui-même ne peut pas s'adapter seul, l'hypothèse de départ émise à la section 4.5.1 est que ce cas particulier n'a pas été prévu lors de sa conception.

La stratégie choisie dans ce cas d'utilisation est la correction dynamique du problème. Par exemple, il serait intéressant d'injecter du comportement pour tracer les activités de l'entité qui reçoit les données du GPS dans *follower-drone*, afin de comprendre le problème. La fonctionnalité de trace est arbitraire : c'est un moyen d'observer le système pour le comprendre. Le contexte du problème étant imprévisible, il est nécessaire de pouvoir construire des vues personnalisées du logiciel en cours d'exécution. Une fois le problème compris, une correction pourrait être expérimentée. Un nouveau comportement, cohérent avec l'analyse préalable du contexte, peut venir adapter le programme pour expérimenter et valider la correction du problème. Une fois la correction validée, le comportement de trace − ou de visualisation personnalisée − doit être retiré du système en cours d'exécution car il n'est plus nécessaire. Les drones pourront alors reprendre leur mission.

Ces deux étapes − visualiser pour investiguer puis corriger le système en cours d'exécution − sont complémentaires, et peuvent être effectuées en parallèle. Les adaptations conçues et injectées pour la visualisation sont à court terme retirées de l'exécutable, tandis que les adaptations de corrections peuvent être capitalisées dans la prochaine version du programme, corrigeant l'erreur de manière durable.

4.5.3 Description de la simulation

La simulation est implémentée avec Pharo Smalltalk. Les drones sont des objets, et tracent leurs actions dans une console. Les deux drones ont la possibilité de communiquer l'un avec l'autre, et de connaître la distance exacte qui les sépare.

Leur comportement d'origine est de se déplacer sur l'axe y d'une grille virtuelle. Cette grille est composée de zones, qui peuvent être *sécurisées* ou *non sécurisées*. Dans le dernier cas, les drones chiffrent leurs communications. Le temps simulé est relatif. Le scénario de cette simulation est une mission durant laquelle le drone *follower-drone* doit utiliser les données GPS du drone *gps-drone* pour se positionner. Sans ces données GPS, *follower-drone* ne peut pas se déplacer et la flotte se bloque dans l'attente d'une solution.

Ce scénario ne dépasse pas ici les bornes d'une simulation, et son passage à l'échelle dans une situation réelle n'est pas étudié. Notamment, nous supposons qu'il n'y a pas d'interruption de communications avec les drones ou entre les drones. Ce type de situation pourrait être géré par des outils existants ou des solutions dédiées pour assurer une communication transparente et ininterrompue avec l'utilisateur-rice. L'étude et la conception de tels outils sort du périmètre de cette évaluation. L'objectif est l'étude de l'adaptation de comportement non-anticipée avec des langages étendus par Lub.

L'utilisation de l'adaptation non-anticipée de comportement pour le déverminage dans ce type de scénario est également limitée par le type d'erreurs que le système rencontre. Une perte totale de communication avec l'un des drones sera difficile à aborder par de l'adaptation dynamique. L'adaptation non-anticipée nécessitant une interaction entre l'appareil et un-e opérateur-rice, l'opération de déverminage en devient compromise.

4.5.4 Adaptations au fil de l'eau

Nous nous focalisons ici sur une partie restreinte du modèle, et qui sera sujette à l'adaptation : la fonctionnalité de communication de drone à drone. Deux classes constituent le cœur de ce modèle : *CommunicationPort* et *LSDrone*.

CommunicationPort. Les instances de *CommunicationPort* modélisent les canaux de communication utilisés par les drones pour transmettre des données. Chaque drone possède une instance de *CommunicationPort* dédiée à la communication avec l'autre drone, et une instance dédiée pour communiquer avec un-e opérateur-rice. Les méthodes *sendMessage*: et *receiveMessage*: permettent respectivement aux drones d'envoyer et de recevoir des messages.

LSDrone. Cette classe modélise les drones de la simulation. Deux instances sont présentes à l'exécution, respectivement *follower-drone* et *gps-drone*. Ces deux objets utilisent leurs instances respectives de *CommunicationPort* pour échanger des coordonnées GPS. Lorsqu'un message est reçu sur le port de communication drone à

drone − c'est-à-dire un message venant d'un autre drone − il est transmis au drone courant via la méthode *handleMessage*: de la classe *LSDrone*.

```
1  "Base class for the simulation drones"
2  Object subclass: #LSDrone
3      handleMessage: aValue
4          | rs xValue yValue |
5          self encryptedMessagesMode
6              ifFalse: [ ^ aValue ].
7          rs := (String fromByteArray: aValue) readStream.
8          xValue := (rs upTo: $.) asNumber.
9          yValue := rs upToEnd asNumber.
10         ^ xValue @ yValue
```

FIGURE 4.20 – Extrait du modèle de la classe *LSDrone*. La méthode *handleMessage*: interprète un message contenant des coordonnées GPS reçues d'un autre drone.

La Figure 4.20 illustre le code de la méthode *handleMessage*:. La méthode prend en paramètre des coordonnées GPS extraites d'un message reçu sur un port de communication. Ces coordonnées sont éventuellement déchiffrées si le drone est entré dans un mode *sécurisé* (lignes 7-10). Si le drone n'est pas entré dans un mode sécurisé, alors il est considéré que les données extraites du message sont directement exploitables (lignes 5 et 6).

Nous pouvons suivre l'exécution du code au travers des traces envoyées par les drones à leur opérateur-rice (Figure 4.21). Au temps simulé t = 5 (ligne 1), *follower-drone* communique avec *gps-drone* pour récupérer des coordonnées GPS et se positionner. Cependant à t = 6 (ligne 5), *follower-drone* entre dans un mode d'attente car il ne peut plus se déplacer. En réponse, *gps-drone* (t = 7, ligne 11) entre également en mode d'attente et la flotte est bloquée. Du point de vue d'un-e opérateur-rice, la situation n'est pas claire : la communication entre les drones est bonne, et la communication externe avec les drones est également fonctionnelle.

Nous pourrions surveiller dans *follower-drone* les données GPS reçues de *gps-drone* pour vérifier leur validité. Cependant ce comportement n'est pas présent dans le système, car cette possibilité n'est pas prévue dans le programme. Il n'est pas possible de savoir quelle entité précise surveiller, ni à quel moment. Ce sont des informations qui ne sont pas connues avant que le problème n'arrive, il est donc nécessaire de pouvoir mettre en place ce genre de surveillance de manière dynamique et non-anticipée.

```
1  t = 5
2  gps-drone moves to : 200@25
3  follower-drone moves to : 100@50
4
5  t = 6
6  gps-drone moves to : 200@50
7  follower-drone moves to : 100@25
8  follower-drone lost its GPS signal
9  follower-drone is in safe mode
10
11 t = 7
12 gps-drone at: 200@75
13 gps-drone is in safe mode
14 follower-drone at: 100@25
15 follower-drone lost its GPS signal
16 follower-drone is in safe mode
```

FIGURE 4.21 – Extrait des traces d'exécution reçues par l'opérateur-rice : à t = 7, la flotte se bloque car le signal GPS est perdu.

Nous proposons d'adapter le comportement du canal de communication drone à drone de *follower-drone* pour surveiller les coordonnées GPS reçues de *gps-drone*. Une classe dédiée implémentant du comportement de trace est développée : il s'agit de la classe *ComPortLogger* de la Figure 4.22. Une première méthode (ligne 2) est créée : *logMessage*:, qui prend en paramètre un message et trace son contenu dans une console. Une seconde méthode (ligne 6), *receiveMessage*:, fait uniquement appel à la méthode *logMessage*: en passant le message reçu en paramètre.

Nous définissons une adaptation basée sur cette classe (Figure 4.23). Cette adaptation est un méta-object nommé *ComPortAdaptation*. Elle spécifie qu'elle adaptera deux méthodes (lignes 3 et 4), *logMessage*: et *receiveMessage*:, venant de la classe *ComPortLogger* (ligne 5). Le contrôle appliqué à la méthode *logMessage*: est du type *acquire*, ce qui signifie que l'objet auquel sera appliquée l'adaptation obtiendra ce nouveau comportement. Le contrôle appliqué à la méthode *receiveMessage*: est du type *before*. Cela signifie que si cette méthode existe déjà dans la classe de l'objet qui sera adapté, alors le comportement adapté sera exécuté *avant* le comportement d'origine, qui sera exécuté à sa suite. Les paramètres éventuels de la méthode d'origine sont passés à la méthode définie par l'adaptation. Si la méthode n'existe pas déjà dans la classe de l'objet adapté, alors le *before* est équivalent à un *acquire*. Enfin, ligne 8, l'adaptation est appliquée à l'instance de *CommunicationPort* du drone *follower-drone* chargée de recevoir les données GPS de *gps-drone*.

```
1  Object subclass: #ComPortLogger
2      logMessage: aMessage
3          "logs a communication message"
4          [... logging code ...]
5
6      receiveMessage: aMessage
7          self logMessage: aMessage
```

FIGURE 4.22 – Adaptation développée pour surveiller les messages reçus : la classe *ComPortLogger* avec une méthode de réception de message et une méthode de trace.

Cet objet sera adapté comme suit :

- L'objet a acquis la méthode *logMessage*:.
- La méthode *receiveMessage*: de l'objet est modifiée comme suit : lors de son exécution, la méthode *receiveMessage*: de *ComPortLogger* sera d'abord exécutée avec le paramètre passé à la méthode d'origine. Cette exécution provoque un appel à la méthode *logMessage*:, qui trace le message passé en paramètre. Puis la méthode *receiveMessage*: de la classe d'origine de l'objet, c'est-à-dire *CommunicationPort*, est exécutée.

Dans les méthodes adaptées ou acquises de l'objet adapté, toutes les références à *self* ou *super* désignent toujours respectivement l'objet lui-même et la super classe de sa classe d'origine. Nous admettons également que l'accès à la référence de l'objet est possible lors de l'exécution. C'est une possibilité nativement offerte par un langage comme Pharo, où l'introspection autorise l'inspection dynamique d'objets et de leur état. Nous admettons dans ce cas que la simulation ouvre un inspecteur Pharo sur la simulation, permettant de parcourir les objets non temporaires (les drones et leurs ports de communication) et d'en obtenir des références auxquelles appliquer des adaptations.

L'adaptation étant désormais active pour le port de communication de *follower-drone*, il est possible d'observer les données GPS reçues de *gps-drone*. Comme la flotte est en attente et ne bouge plus, les coordonnées attendues sont celles de la dernière position connue de *gps-drone*, soit *200@75*. Les traces d'exécution de la Figure 4.24 montrent les nouvelles informations produites par l'adaptation du port de communication drone à drone de *follower-drone*. Aucune des autres instances de *CommunicationPort*, c'est-à-dire celles de *gps-drone* et le port drone-opérateur-rice de *follower-drone*, n'est affectée par l'adaptation − il n'y a pas de traces supplémentaires pour ces instances. Seule la méthode spécifiée par l'adaptation, c'est-à-dire la méthode *receiveMessage*:, est altérée par la modification comportementale.

```
1  Adaptation
2      named: 'ComPortAdaptation'
3      adapt: {#receiveMessage: -> #before.
4              #logMessage: -> #acquire}
5      with: ComPortLogger.
6
7  followerDrone peerDroneComPort
8      adaptWith: (Adaptation named: 'ComPortAdaptation')
```

FIGURE 4.23 – Adaptation *ComPortAdaptation*, définissant des variations comportementales qui trouvent leur source dans la classe *ComPortLogger*. L'adaptation est appliquée à l'instance de *CommunicationPort* du drone *follower-drone* chargée de recevoir les données GPS de *gps-drone*.

Dans ce scénario, l'opérateur ou l'opératrice réalise que les données reçues sont chiffrées (lignes 6 et 7), car *gps-drone* est entré dans une zone dite non sécurisée. L'autre drone n'a pas été notifié de ce changement de contexte, et ne sait pas qu'il doit désormais déchiffrer les données qu'il reçoit. Il s'agit d'un *bug*, car *follower-drone* suit le premier drone et n'est donc pas encore entré dans la zone non sécurisée. Il n'a donc ni activé son mode "*non sécurisé*", ni reçu de notification car ce mécanisme n'existe pas dans la conception originale du programme.

```
1  t = 10
2  gps-drone at: 200@75
3  gps-drone is in safe mode
4
5  ### follower-drone receiving message: pinpointAnswer
6  ### contents: #[50 48 48 46 55 53]
7  follower-drone lost its GPS signal
8  follower-drone is in safe mode
9  follower-drone at: 100@25
```

FIGURE 4.24 – Traces d'exécution après adaptation : le port de communication drone à drone de *follower-drone* est le seul objet affecté par l'adaptation comportementale et trace de nouvelles informations (lignes 5 et 6).

Pour adapter *follower-drone* à ce contexte, il faut que ce dernier puisse détecter si les données reçues sont chiffrées. Une adaptation possible est de vérifier le format des données GPS reçues passées à *follower-drone* dans la méthode *handleMessage*:

(Figure 4.20). Nous définissons une classe *LSDroneWithEncryptionMode* qui implémente la méthode *handleMessage*: (Figure 4.25). Dans cette version, la validité des données GPS reçues est vérifiée – ici il est juste vérifié s'il s'agit d'un objet de la classe *Point* (ligne 3), et si ce n'est pas le cas, les données sont probablement chiffrées. Dans le cas où les données seraient invalides, le drone force le passage en mode "*non sécurisé*" (ligne 4).

```
1 Object subclass: #LSDroneWithEncryptionMode
2     handleMessage: aValue
3         aValue isPoint
4             ifFalse:[self encryptedMessagesMode: true]
```

FIGURE 4.25 – Définition dans une nouvelle classe de la méthode *handleMessage*:, qui vérifie la validité des données passées en paramètre. Si ces dernières ne correspondent pas à un point, alors le drone entre dans le mode "*non sécurisé*".

Une nouvelle adaptation est définie : *DataValidationAdaptation* (Figure 4.26). Cette dernière spécifie une adaptation de la méthode *handleMessage*: (ligne 2) qui se situe dans la classe *DataValidationAdaptation* (ligne 3). Le contrôle de type *before* (ligne 2) précise qu'à l'exécution de la méthode *handleMessage*: d'un objet adapté, la méthode de la nouvelle classe *DataValidationAdaptation* sera d'abord exécutée, suivie de l'exécution de la méthode originale de l'objet adapté. Enfin, l'adaptation est appliquée au drone *follower-drone* (ligne 5). Comme précédemment, les références manipulées (ici *followerDrone*) peuvent être facilement retrouvées avec les outils de Pharo, et sont présentées comme étant dynamiquement trouvées à l'exécution avec l'inspecteur de Pharo.

```
1 Adaptation named: 'DataValidationAdaptation'
2         adapt: {#handleMessage: -> #before}
3         with: LSDroneWithEncryptionMode
4
5 followerDrone adaptWith:
6     (Adaptation named: 'DataValidationAdaptation')
```

FIGURE 4.26 – Adaptation de la méthode *handleMessage*: du drone *follower-drone*, par la méthode *handleMessage*: de la classe *LSDroneWithEncryptionMode*.

L'objet *follower-drone* est désormais adapté avec l'adaptation *DataValidationAdaptation*. Lors de l'exécution, la méthode *handleMessage*:

de *follower-drone* est donc modifiée comme suit :

- La méthode *handleMessage*: de la classe $LSDroneWithEncryptionMode$ (Figure 4.25) sera d'abord exécutée. Cette dernière vérifie si les données fournies en paramètre correspondent bien à celles d'un point. Si tel n'est pas le cas, l'objet courant (correspondant au drone) passe en mode "*non sécurisé*".
- La méthode d'origine *handleMessage*: (Figure 4.20) s'exécute ensuite. Si le drone est passé en mode "*non sécurisé*", les données seront déchiffrées avant le paramétrage de la position.

Le résultat de l'adaptation est visible dans les traces transmises à l'opérateur-rice lors de l'exécution (Figure 4.27). Au temps t = 12 (ligne 1), *follower-drone* reçoit à nouveau les mêmes coordonnées chiffrées (ligne 5), mais se déplace à nouveau (ligne 6). À t = 13 (ligne 8), la flotte a repris sa mission et les deux drones se déplacent (lignes 9 et 12). La ligne 11 montre les nouvelles coordonnées chiffrées reçues par *follower-drone* qui ont été mises à jour par *gpsDrone*. Nous pouvons en déduire que l'adaptation a permis de forcer le passage en mode "*non sécurisé*" de *follower-drone*, qui a pu déchiffrer les coordonnées GPS reçues de *gpsDrone*. Les deux drones ont repris leur mission car la situation bloquante a été résolue.

```
1  t = 12
2  gps-drone at: 200@75
3  gps-drone is in safe mode
4  ### follower-drone receiving message: pinpointAnswer
5  ### contents: #[50 48 48 46 55 53]
6  follower-drone at: 100@50
7
8  t = 13
9  gps-drone moves to : 200@100
10 ### follower-drone receiving message: pinpointAnswer
11 ### contents: #[50 48 48 46 49 48 48]
12 follower-drone moves to : 100@75
```

FIGURE 4.27 – Résultat de l'exécution après adaptation : les drones reprennent leur route car *follower-drone* a pu déchiffrer les coordonnées GPS de *gpsDrone*.

Cependant, le port de communication drone à drone de *follower-drone* continue à tracer les données GPS reçues de *gps-drone* (Figure 4.28). Il s'agit d'une adaptation conçue et mise en place pour comprendre le problème. Son objectif atteint, cette adaptation n'a pas de raison de persister dans le programme en cours d'exécution. L'instance adaptée de *CommunicationPort* doit retrouver son

comportement d'origine. Les lignes 1 et 2 du code présenté par la Figure 4.28 suppriment le comportement adapté de l'objet *CommunicationPort*, tandis que les lignes 4 à 7 montrent le résultat des traces. Les traces ajoutées par l'adaptation ne sont plus présentes.

```
1 followerDrone peerDroneComPort revertAdaptation:
2     (Adaptation named: 'ComPortAdaptation')
3
4 "Log results"
5 t = 14
6 gps-drone moves to : 200@100
7 follower-drone moves to : 100@75
```

FIGURE 4.28 – L'adaptation du port de communication drone à drone de *follower-drone* est retirée du programme car elle n'est plus nécessaire.

4.5.5 Discussion

Nous avons présenté un scénario dans lequel une simulation d'une flotte de drones rencontre un problème dû à un contexte non-anticipé lors de la conception du système. Des adaptations dynamiques et non-anticipées de comportements sont mises en place, d'abord pour sonder le système puis pour le corriger. Ces adaptations s'appliquent à des objets uniques, sans affecter les autres instances de même classe que les objets adaptés. C'est un cas typique d'adaptation non-anticipée : le problème est clairement dû à un *bug*, qui est, par définition, imprévu lorsqu'il se produit pendant l'exécution du programme. Avant qu'il ne se produise, et qu'il ne soit compris et corrigé, les adaptations qui seront mises en place restent inconnues.

Tout au long de ces adaptations, nous pouvons observer que l'état des objets est préservé. Par exemple, les drones possèdent toujours les mêmes noms et les mêmes coordonnées avant et après les adaptations. Les références et l'identité ($self$) des objets sont également préservées. Ceci est implicitement visible dans les traces : les ports de communication de chaque drone, instances de *CommunicationPort*, sont toujours connectés à leurs destinataires pour envoyer et recevoir des messages.

Les adaptations affectent le code en cours d'exécution de manière dite *non intrusive*. Cela signifie que :

— Le code d'origine d'un objet est préservé : le comportement d'un objet est remplacé dynamiquement par un comportement adapté, mais il reste présent et il est toujours possible de revenir au comportement d'origine du système.

— Le code du programme n'est pas modifié : si un dévermineur s'ouvre sur une exception ou un point d'arrêt dans une méthode adaptée, aucune instrumentation n'est visible dans le code présenté.

Cela signifie également que, dans un cas comme celui présenté dans cette section, une adaptation corrigeant un problème n'est que temporaire. Une solution pérenne doit être développée avec des moyens plus conventionnels, puis le programme des drones doit être mis à jour et déployé lors de la prochaine mission (ou la prochaine simulation). L'adaptation non-anticipée reste importante, car elle permet de mettre au point, voire de corriger un programme dont l'exécution ne doit pas s'interrompre. Des solutions automatiques de génération de correctifs intégrables dans le code source pourraient être étudiées, mais sortent du périmètre de la thèse.

Dans le scénario présenté, une adaptation de comportement est appliquée sur un objet pour produire des traces, puis le comportement d'origine de l'objet est restauré. Il est important de noter que lors d'une restauration de comportement, qui peut impliquer plusieurs méthodes, il peut y avoir des incohérences dans l'état de l'objet. Un comportement adapté peut faire usage d'une variable d'instance de l'objet et la modifier sans garantie d'en maintenir un état cohérent. Ce problème devrait idéalement être géré par le mécanisme d'adaptation sous-jacent, qui validerait les adaptations. À défaut, il faut manuellement s'assurer, lors de la conception d'adaptation, que le nouveau comportement ne sera pas source d'incohérences après restauration de l'objet.

La compréhension d'erreurs est facilitée par l'accès au contexte du programme en cours d'exécution. Sonder et expérimenter au travers d'adaptations de comportement permet d'observer l'environnement du programme, qui peut être très complexe. Sans cette possibilité d'accès au contexte au moment d'un problème, trouver et comprendre toutes les conditions nécessaires pour reproduire une erreur peut s'avérer très difficile. Cela n'est cependant pas trivial de trouver un point d'entrée dans un système à instrumenter. Il faut savoir quel objet observer au bon moment, et quel objet adapter pour expérimenter un correctif. De très nombreux objets peuvent exister au cours du fonctionnement d'un programme, et certains peuvent avoir une durée de vie extrêmement limitée dans le temps. Choisir les objets à adapter, ou bien savoir comment concevoir une adaptation, repose pleinement sur la compréhension du programme par le ou la développeur-euse, et dépend de son interprétation des informations fournies par le contexte du programme en cours d'exécution.

Enfin, le déverminage par le biais d'adaptation non-anticipée ne peut pas s'appliquer à tous les types de problèmes. C'est le cas, par exemple, quand l'application est stoppée à cause d'une erreur et que le programme est arrêté. C'est également le cas des problèmes qui affectent la communication avec le dévermineur. Les exceptions non gérées par le programme peuvent provoquer des comportements instables si elles ne sont pas redirigées vers un dévermineur hypothétique. Ce type de problème peut empêcher l'utilisation de mécanismes d'adaptation non-anticipée.

4.6 Évaluation

La table 4.6 présente l'évaluation de la mise en œuvre de Lub avec Pharo, au regard des propriétés recherchées pour l'adaptation non-anticipée d'objets. L'extension de Reflectivity est également positionnée, satisfaisant désormais la propriété d'adaptation à granularité objet grâce à ses metalinks centrés objet. Dans ce qui suit, cette mise en œuvre avec Pharo sera simplement désignée par Lub.

Nous avons montré que Lub était capable d'adapter un objet unique, sans affecter les autres instances de la même classe que cet objet (Figure 4.24). La propriété d'adaptation avec une granularité objet est donc satisfaite.

L'adaptation est appliquée directement sur un objet. Elle peut être à tout moment retirée de cet objet pour que ce dernier retrouve son comportement original, tel que défini par sa classe. L'API fournie par Lub (Figure 4.18) permet ces activations et désactivations dynamiques d'une adaptation en envoyant directement les messages correspondants à l'objet ciblé. La propriété de réversibilité est donc satisfaite.

Lub est basé sur l'extension de Reflectivity que nous avons développée, et utilise ses metalinks centrés objet pour fournir des adaptations à granularité objet. L'utilisateur-rice n'a cependant pas à manipuler de concepts proches du langage ou de la structure du programme, comme le *lookup* ou l'AST. L'utilisation même de Reflectivity et des metalinks est masquée par l'infrastructure de Lub, et l'utilisateur n'a à se soucier que de la sémantique de l'adaptation. La conception d'un comportement adapté se fait par la création ou la réutilisation d'une classe (Figure 4.22), et la déclaration d'une adaptation (Figure 4.23). La définition du comportement adapté utilise donc les mêmes concepts (les classes) et les mêmes moyens (les méthodes) que la définition du comportement standard du programme. Seul le concept d'adaptation est nouveau dans le langage, mais ne nécessite qu'une simple déclaration spécifiant une classe cible et le nom des méthodes qui implémentent le comportement adapté. En outre, l'application de l'adaptation est contrôlée exclusivement au travers de l'API de Lub (sept méthodes). Nous considérons alors la propriété de flexibilité comme atteinte.

Cependant, si Lub peut individuellement adapter des objets, il n'y a aucun moyen pratique d'obtenir des références à ces objets. C'est à l'utilisateur-rice de les trouver manuellement et d'utiliser l'interface de Lub pour les adapter. Si l'obtention de références à des objets dont la durée de vie est stable durant toute l'exécution du programme est réalisable, il en est tout autrement pour les objets dont la durée de vie est extrêmement courte, ou qui ne sont pas référencés explicitement par des variables. En conséquence, la propriété de minage d'objet n'est pas satisfaite.

Finalement, les propriétés de préservation d'identité, de non-intrusivité et du maintient du paradigme objet sont hérités de Reflectivity, composante essentielle

sur laquelle repose la mise en œuvre de Lub dans Pharo.

● Totalement supporté ◑ Partiellement supporté ○ Non supporté

	Granularité objet	Minage d'objets	Réversibilité	Identité préservée	Flexibilité	Non-intrusivité	Paradigme objet
Reflectivity	○	○	◑	●	◑	●	●
Reflectivity + extension	●	○	◑	●	◑	●	●
Lub	●	○	●	●	●	●	●

TABLE 4.6 – Positionnement de Lub au regard de nos contraintes pour l'adaptation non-anticipée.

4.7 Conclusion

Dans ce chapitre, nous avons décrit des éléments de mise en œuvre généraux permettant d'étendre un langage objet à typage dynamique à l'adaptation non-anticipée de comportement, telle que décrite par le patron de langage Lub. Deux mises en œuvre concrètes sont évoquées, une avec le langage Pharo et une avec le langage Python. Les détails d'implémentation de la mise en œuvre avec Pharo sont présentés et notamment au travers d'une extension de Reflectivity, qui fait partie de la couche réflexive du langage. Cette extension permet l'application d'opérations réflexives à des objets uniques, et sera officiellement intégrée dans une prochaine version du langage. L'extension de Reflectivity est utilisée comme mécanisme d'adaptation primitif par Lub, pour mettre en œuvre l'adaptation non-anticipée de comportement. L'API de Lub apporte des facilités syntaxiques pour l'utilisation des capacités d'adaptation et leur application à des objets précis. L'utilisation de Lub reste strictement dans le périmètre du paradigme objet, et l'utilisateur-rice manipule toujours des concepts d'objets et de classes. Le concept d'adaptation est ajouté, mais se limite à la manipulation des concepts objets.

Une analyse de l'impact sur les performances et sur la mémoire du système est présentée pour la mise en œuvre avec Pharo. Deux manières d'adapter son comparées : l'adaptation par les mécanismes objets natifs de Pharo et l'adaptation avec Lub. Cette évaluation mesure les différences en terme de temps d'exécution du lookup, de temps d'exécution nécessaire à l'adaptation ainsi que de l'espace mémoire

supplémentaire occupé. Une analyse critique des résultats est proposée. Dans les cas simples étudiés, le coût de l'adaptation n'est pas nul. Un système avec des ressources limitées pourrait être perturbé par l'adaptation non-anticipée, ou bien celle-ci ne pourrait être que chirurgicale – ce qui est un des objectifs principaux de Lub. La responsabilité d'appliquer de telles adaptations revient finalement à l'utilisateur-rice, qui doit décider si le coût de l'adaptation est abordable au regard des possibilités qu'elle apporte.

Un cas d'utilisation d'un problème dans une simulation d'une flottille de drones est présenté. L'adaptation non-anticipée d'une instance du modèle permet d'abord de sonder le système en cours d'exécution pour comprendre le problème, via un système de traces. Une fois le problème compris, une adaptation de comportement est appliquée à un drone pour corriger l'erreur. Le problème résolu, l'adaptation générant des traces pour sonder le programme est désactivée et retirée de l'exécution. Cette évaluation montre l'utilisation de l'adaptation non-anticipée de comportement avec Lub pour déverminer un programme. Une discussion de cette évaluation est finalement proposée.

Enfin, nous avons évalué la mise en œuvre Pharo de Lub au regard des propriétés recherchées pour l'adaptation non-anticipée de comportement, qui sont toutes satisfaites sauf la propriété de minage d'objet.

Chapitre 5

Mise en œuvre du patron de langage pour la collecte d'objets non-anticipée : les Collecteurs

5.1 Contraintes et options de mise en œuvre des Collecteurs 161
5.1.1 Contraintes de mise en œuvre 161
5.1.2 Options de mise en œuvre pour l'application au déverminage . 163
5.1.3 *Metalinks* dédiés pour la collecte d'objets 165
5.1.4 Collecte d'objets *en profondeur* 166
5.2 Mise en œuvre concrète des Collecteurs 166
5.2.1 Modèle de mise en œuvre avec Pharo et Reflectivity 166
5.2.2 Limites du modèle 172
5.2.3 L'API des Collecteurs 172
5.2.4 Mise en œuvre avec Python et Reflectivity 176
5.3 Outils pour l'application au déverminage 177
5.3.1 Outillage des Collecteurs 178
5.3.2 Visualisation des objets collectés : les historiques de collecte . 181
5.4 Traces d'objets non-anticipées dans un programme en cours d'exécution 184
5.4.1 Description du problème 184
5.4.2 Traces dynamiques d'objets collectés 185
5.4.3 Discussion 185
5.5 *Record/Replay* pour la reproduction de valeurs non-déterministes . . 186
5.5.1 Description du problème 186
5.5.2 Rejouer une exécution avec un objet collecté 187
5.5.3 Discussion 191
5.6 Historiques de collecte pour l'investigation de problèmes difficiles . . . 192
5.6.1 Description du problème 193

5.6.2 Observation du programme avant la modification problématique 194
5.6.3 Observation du programme après introduction de l'erreur . . . 195
5.6.4 Traque du problème et correction 196
5.6.5 Discussion 199
5.7 Discussion générale et évaluation 200
5.7.1 Évaluation 200
5.7.2 Discussion 201
5.8 Conclusion 202

Dans le chapitre 3, nous avons décrit Lub et les Collecteurs dans un patron de langage pour l'adaptation non-anticipée du comportement d'objets. Dans le chapitre 4, nous avons décrit et évalué la mise en œuvre de Lub dans Pharo, qui permet d'adapter le comportement d'objets dans un programme en cours d'exécution. Dans ce chapitre, nous décrivons les principes et les techniques de mise en œuvre des Collecteurs, qui permettent de spécifier et de récupérer les objets qui seront adaptés par Lub. Cette opération est nommée *collecte* d'objets non-anticipée, et vise à satisfaire la propriété de *minage d'objets* décrite dans le chapitre 1.

Nous présentons une mise œuvre des Collecteurs dans Pharo avec Reflectivity et son extension (décrite dans le chapitre 4). Cette mise en œuvre est constituée d'un modèle des Collecteurs, d'une interface utilisateur et d'un outillage développé à l'attention des développeur-euse-s. Les Collecteurs constituent un patron de langage indépendant de Lub et de l'adaptation de comportement. Indépendamment de l'adaptation, nous utilisons donc les collecteurs pour proposer des outils dédiés au déverminage de programmes à objets, qui mettent en œuvre des techniques permettant de rejouer l'exécution d'une partie d'un programme à partir d'objets collectés.

Nous évaluons les Collecteurs au travers de trois cas d'utilisation. Le premier cas d'utilisation illustre la possibilité d'injecter des traces d'exécution dans un programme en cours d'exécution. Les Collecteurs permettent de miner des objets à partir du flot de contrôle du programme, et de filtrer ces objets par des conditions (ou gardes) pour la production de traces. La spécification des objets à collecter, des conditions de collecte et du comportement de traces sont complètement non-anticipées. Le second cas d'utilisation décrit un problème dû à une valeur non-déterministe. Nous montrons comment les collecteurs permettent de capturer cette valeur puis de la rejouer pour éliminer l'aspect non-déterministe de l'exécution et corriger le problème. Le dernier cas d'utilisation est une session de déverminage d'un outil nommé *Pillar*, dans lequel l'ajout d'un accesseur provoque l'échec de tests unitaires sans aucun rapport avec la modification. Cette expérimentation met en avant *l'historique de collecte* fournit par les Collecteurs, et qui permet de remonter dans le temps l'exécution d'une expression ciblée par un point de collecte.

Ces travaux sur les Collecteurs ont donné lieu à deux publications [Costiou et al., 2018b, Costiou et al., 2018c].

5.1 Contraintes et options de mise en œuvre des Collecteurs

Les Collecteurs définissent des groupes d'objets dynamiques, qui sont collectés pendant l'exécution du programme. Ces objets deviennent alors accessibles au reste du programme, l'objectif étant de pouvoir leur appliquer des opérations de déverminage comme par exemple des adaptations de comportement. Dans cette section, nous définissons les principes de mise en œuvre des *Collecteurs*. Ces principes constituent des guides de mise en œuvre pouvant être appliqués à tous les langages décrits par *Kernel-Lub*. Des contraintes de mise en œuvre sont énumérées au niveau langage (section 5.1.1), pour que ce dernier soit éligible à une extension par les Collecteurs. Nous définissons également des options de mise en œuvre (section 5.1.2), qui sont implémentées pour exploiter les possibilités des Collecteurs au travers d'outils pour le déverminage. Pour implémenter les Collecteurs, nous reposons sur Reflectivity et son extension, décrite à la section 4.2. Nous explicitons alors comment Reflectivity aborde les contraintes et les options pour la mise en œuvre des Collecteurs.

5.1.1 Contraintes de mise en œuvre

L'objectif des collecteurs est la récupération d'objets à partir d'expressions précises lors de l'exécution du programme. La Figure 5.1 illustre l'opération de collecte. Le point de collecte est symbolisé par l'encadré en pointillé autour de l'expression ciblée (à gauche). À l'exécution, l'évaluation de cette expression précise est suivie de la collecte de son résultat (à droite). L'objet collecté est stocké dans la mémoire du programme, selon la stratégie de gestion mémoire choisie – par défaut la conservation des objets par copies.

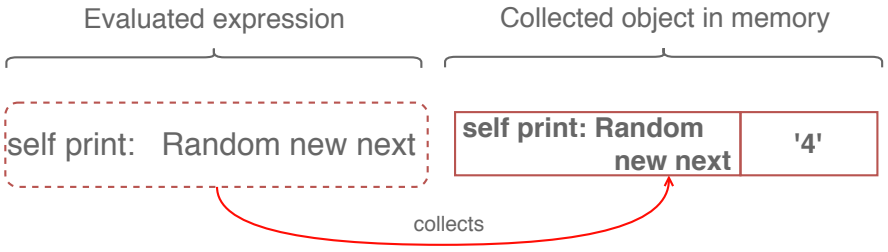


FIGURE 5.1 – Un collecteur : l'encadré en pointillé représente le point de collecte (à gauche), dont le résultat de l'évaluation (un objet) est collecté à l'exécution. Le résultat en mémoire est visible à droite : la case de gauche représente l'expression instrumentée et la case de droite la valeur collectée qui annote cette expression.

En terme de mise en œuvre, le concept de collecteur est réalisable par l'instrumentation directe du code source. Cette instrumentation est mise en œuvre par la ré-écriture des instructions ciblées par des points de collecte dans les méthodes pour lesquelles des collecteurs sont définis. La ré-écriture *extrait* des parties de certaines expressions et assigne leur résultat dans des variables temporaires, puis injecte l'instruction de collection. Un tel exemple de ré-écriture est illustré par les figures 5.2 et 5.3. Dans l'exemple de code en *Smalltalk* de la Figure 5.2, nous souhaitons collecter le résultat de l'expression *Random new next*. Cette expression est évaluée en paramètre du message *print*: envoyée à *self*.

```
1 self print: Random new next.
```

FIGURE 5.2 – Le résultat de l'expression *Random new next*, paramètre de la méthode *print*:, doit être collecté.

Le résultat de la ré-écriture du code de la Figure 5.2 pour l'installation d'un collecteur est visible dans la Figure 5.3. Cette ré-écriture suit les étapes suivantes :

1. Extraction de l'expression à instrumenter (*Random new next*) et affectation de son résultat dans une variable temporaire *tmp*1. Les variables temporaires injectées lors de la ré-écriture de code doivent porter des noms uniques, afin d'éviter les conflits entre expressions instrumentées.
2. Appel du collecteur − référencé ici par une variable nommée *theCollector* − et appel de l'instruction de collecte avec comme paramètre le résultat de l'expression évaluée, stockée dans la variable temporaire *tmp*1.
3. Remplacement de l'expression d'origine en paramètre de la méthode *print*: par la lecture de la variable temporaire *tmp*1 dans laquelle le résultat de l'évaluation de cette expression aura été assigné.

Contrainte de mise en œuvre n^{o}1

Le langage hôte doit permettre la ré-écriture et la recompilation du code source d'une méthode pendant l'exécution du programme.

La contrainte de mise en œuvre n^{o}1 est corollaire de la contrainte n^{o}1 définie pour l'implémentation de Lub (section 4.1.1). Si le code source d'une méthode est ré-écrit, il faut pouvoir soit recompiler cette méthode à la volée, soit charger dynamiquement sa nouvelle version pendant l'exécution du programme. Il s'agit de la

```
1 tmp1 := Random new next.
2 theCollector collect: tmp1.
3 self print: tmp1.
```

FIGURE 5.3 – Ré-écriture de code source pour l'installation d'un collecteur : l'expression instrumentée *Random new next* est extraite de sa position d'origine dans le code de la méthode et assignée dans une variable temporaire *tmp*1. L'instruction de collection est injectée pour la récupération de l'objet résultant de l'évaluation de l'expression d'origine. La variable *tmp*1 remplace ensuite l'expression d'origine en tant que paramètre de la méthode *print*:.

seule contrainte, nécessaire et suffisante, pour mettre en œuvre le patron des Collecteurs dans une version naïve. Toute autre option relève du détail d'implémentation, pour être conforme à leur sémantique opérationnelle (informellement décrite dans le chapitre 3) ou pour développer de l'outillage.

5.1.2 Options de mise en œuvre pour l'application au déverminage

Les Collecteurs représentent un patron d'extension de langage objet dynamique à part entière. Leur sémantique opérationnelle permet d'envisager des mises en œuvre variées pour le déverminage de programmes à objets, et notamment en ce qui concerne l'outillage des Collecteurs. Dans cette section, nous définissons des options d'implémentation que nous utilisons pour produire des outils de déverminage exploitant les possibilités de la sémantique opérationnelle informelle des Collecteurs décrite dans le chapitre 3.

Collecteurs non intrusifs. Dans les faits, pour que la collecte du résultat d'une évaluation soit possible, il faut que le code qui s'exécute soit transformé (section 5.1.1). Rendre ces modifications *non-intrusives* consiste à les rendre transparentes dans le code source, que cela soit dans un environnement déconnecté de l'exécutable ou dans un environnement interagissant avec ce dernier (par exemple un déverminer). Lors d'une opération de déverminage, par exemple si le programme s'arrête à un point d'arrêt dans une méthode qui a été ré-écrite, le développeur ou la développeuse visualisera et interagira avec le code source d'origine et non pas avec le code instrumenté. Cette *transparence* s'applique à l'utilisateur-rice et à l'outillage, avec lequel le code instrumenté en cours d'exécution restera compatible.

Option de mise en œuvre n^{o}1

L'injection d'instrumentations doit être non intrusif, et la modification du code source du programme est transparente du point de vue utilisateur.

Collecteurs centrés objet. Un collecteur agrège des objets issus de l'évaluation d'expressions ciblées dans la syntaxe abstraite du programme − ce sont les points de collecte. Ces points de collecte sont définis dans des méthodes, donc avec un grain qui se situe au niveau de la classe. Obtenir un grain plus fin que la classe est une option alléchante dans le cadre du déverminage de programmes à objet.

Option de mise en œuvre n^{o}2

Un collecteur peut être défini pour un objet particulier : l'opération de collecte ne s'effectue que lorsque l'objet voulu exécute le code contenant l'expression ciblée par un point de collecte.

Collection de méta-informations. Le contexte d'un point de collecte est constitué de nombreuses méta-informations non accessibles pour l'utilisateur-rice. Associer ces méta-informations à la collecte d'un objet apporte de nombreux détails sur l'état de l'objet et de l'expression évaluée. Ces méta-informations peuvent être, par exemple, la pile d'exécution qui indique comment le programme en est arrivé à l'exécution d'un point de collecte donné, ou encore le nom de la variable dans laquelle l'objet est stocké.

Option de mise en œuvre n^{o}3

Les méta-informations d'un point de collecte peuvent être réifiées lors de la collecte d'un objet, afin de lui associer un contexte d'exécution.

Collection d'objets en profondeur. Un point de collecte est une expression. Le corps de cette expression peut éventuellement être constitué de sous-expressions (Figure 3.10, section 3.3). L'évaluation de chaque sous-expression produit un objet, alors utilisé pour l'évaluation de la sous-expression suivante dans le flot de contrôle. C'est le cas dans la Figure 5.2, où le message *print*: prend en paramètre l'objet résultant de l'évaluation de l'expression *Random new next*. L'objet issu de l'évaluation de la sous-expression *Random new* est à ce moment définitivement perdu, car transformé suite à la réception du message *next*. Un collecteur installé sur l'expression complète *self print*: *Random new next* pourrait donc également s'installer

sur ses sous-expressions. Lors d'une opération de déverminage, si le résultat de la collecte globale est suspicieux, il sera alors possible d'observer les objets collectés pour chaque sous-expression.

Option de mise en œuvre n^{o}4

Un collecteur peut s'installer *en profondeur* sur un point de collecte. L'objet résultant de l'évaluation du point de collecte sera collecté, ainsi que les objets issus de l'évaluation des sous-expressions constituant le point de collecte.

5.1.3 *Metalinks* dédiés pour la collecte d'objets

Les metalinks sont des annotations de l'arbre de syntaxe abstraite du langage (AST) fournis par Reflectivity (section 4.2). Les metalinks exécutent du comportement spécifique lors de l'évaluation du nœud sur lequel ils sont installés. Combinés avec les metalinks centrés objets, ils permettent de satisfaire à la contrainte de mise en œuvre n^{o}1 ainsi qu'aux options n^{o}1, n^{o}2 et n^{o}3.

Contrainte n^{o}1 : Réécriture de l'AST d'une méthode pour la collecte. Un nœud annoté par un metalink se comportera de manière similaire à l'exécution. Les objets, valeurs ou concepts que le metalink va *extraire* de l'exécution du nœud seront dynamiquement stockés dans des variables temporaires. Ces entités extraites par le *metalink* sont des *réifications* du contexte d'exécution du nœud. Ces réifications sont ensuite accessibles par le metalink pour être utilisées dans le code qu'il exécute. Les metalinks peuvent donc produire un comportement similaire à celui décrit par la Figure 5.3, et nous permettent d'implémenter la collecte d'objet. L'objet collecté correspond alors à une réification particulière du nœud (la *valeur* produite).

Option n^{o}1 : Instrumentations de collecte non intrusives. L'utilisation de Reflectivity permet immédiatement de rendre les instrumentations non intrusives. Les annotations des nœuds de l'AST ne sont pas visibles dans le code source, et les variables temporaires utilisées pour stocker et fournir les réifications sont générées dynamiquement pour l'exécution de la méthode instrumentée.

Option n^{o}2 : Collecteurs centrés objet. L'utilisation des metalinks centrés objet (section 4.2.2.1) permettent immédiatement de restreindre le périmètre d'une collecte à un objet. L'instrumentation mettant en œuvre la collecte de l'objet issu de l'évaluation d'une expression est alors spécifique au flot de contrôle d'une instance spécifique.

Option n°3 : Collecte de méta-informations. La configuration de metalinks permet de requérir certaines réifications du contexte d'exécution d'un nœud. Outre ces réifications, la conception du méta-objet contenant le comportement qui sera exécuté par le metalink reste libre. Il est donc possible d'implémenter pour un metalink des méta-objets qui minent des informations spécifiques dans le contexte du point de collecte, par exemple le receveur d'un message ou le nom d'une variable.

5.1.4 Collecte d'objets *en profondeur*

La collection d'objets en profondeur (option n°4) consiste à collecter tous les objets issus de l'évaluation des sous-expressions du point de collecte. C'est une stratégie (optionnelle) d'installation du collecteur sur un nœud de l'AST, et une gestion particulière du point de collecte ainsi défini. Lors de l'installation du collecteur, ce dernier parcourt l'arbre du nœud cible et tente de s'installer sur chacun de ses nœuds fils ("en profondeur"). La Figure 5.4 montre l'expression de la Figure 5.2 instrumentée par une collection en profondeur. À l'exécution, le résultat de chaque sous-expression est collecté et stocké. L'ordre de la collecte dépend de l'ordre d'évaluation des sous-expressions. Tous les objets collectés dans les sous-expressions sont liés au point de collecte ciblé par le collecteur, c'est-à-dire ici l'expression principale. Le paramétrage du point de collecte est appliqué pour toutes les instrumentations des sous-expressions. Par exemple, si un collecteur est centré objet et requiert un certain nombre de réifications du contexte, toutes les sous-instrumentations du point de collecte seront également centrées objet et requerront les mêmes réifications. Toutes ces instrumentations sont désinstallées lorsque le collecteur lui-même est désinstallé, ou bien que son point de collecte est mis à jour − dans ce cas l'installation en profondeur est à nouveau effectuée sur ce nouveau point de collecte.

5.2 Mise en œuvre concrète des Collecteurs

Les Collecteurs sont d'abord implémentés dans le langage Pharo suivant les techniques présentées par les sections 5.1.3 et 5.1.4. Dans cette section, le modèle de mise en œuvre reposant sur ces techniques est explicité. Une interface d'utilisation des Collecteurs reposant sur ce modèle est ensuite décrite sous la forme d'une API. Les Collecteurs sont également mis en œuvre dans le langage Python suivant les mêmes techniques et les mêmes modèles.

5.2.1 Modèle de mise en œuvre avec Pharo et Reflectivity

Le modèle d'implémentation des Collecteurs est illustré par la Figure 5.7. Il repose sur des éléments natifs de Pharo, respectivement les classes *Object*,

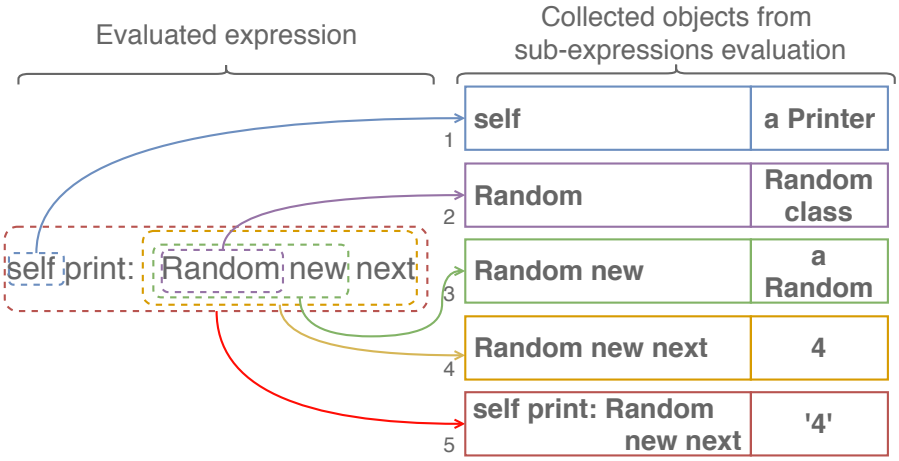


FIGURE 5.4 – Collection en profondeur. Le point de collecte est l'expression globale. Le collecteur s'installe sur chaque sous-expression du point de collecte (à gauche), et les objets issus de leur évaluation sont collectés (à droite).

RBProgramNode et *MetaLink*.

Object. Représente un objet quelconque en mémoire. Il s'agit de la super classe abstraite de tous les objets du langage. Elle n'est pas instanciée en tant que telle, mais représente une abstraction des objets collectés du point de vue d'un collecteur.

RBProgramNode. Représente un nœud de l'arbre de syntaxe abstraite du langage (AST). Ce nœud représente une expression quelconque du programme. Ses sous-classes spécialisées représentent les éléments concrets de la syntaxe abstraite (variables, méthodes...).

MetaLink. Les instances de MetaLink jouent le rôle d'annotations des nœuds de l'AST. La relation entre les metalinks et les nœuds annotés est ici simplifiée. Du point de vue d'un collecteur un metalink annote un ou plusieurs nœuds, mais un même noeud ne peut pas être annoté plus d'une fois par le même metalink.

Le modèle est ensuite constitué des classes formant la mise en œuvre des Collecteurs. Ces éléments et leurs interactions sont décrits ci-après.

Collector. Un collecteur, instance de *Collector*, accumule des objets issus de l'évaluation d'expressions cibles pendant l'exécution du programme. Ces expressions sont ciblées par des points de collecte, définis par le collecteur. Un collecteur est unique,

et doit être nommé. Les collecteurs ne peuvent pas être composés, et sont indépendants les uns des autres du point de vue de la collecte. L'installation d'un collecteur peut être effectuée avant l'exécution du programme, ou de manière non-anticipée pendant son exécution.

Le collecteur prédéfinit un metalink de collecte, associé à son point de collecte, qui déclenche le comportement de collecte. Ce metalink est installé sur le nœud de l'AST ciblé par un point de collecte, et également sur tous les sous-nœuds contenus dans cet AST dans le cas de la collecte en profondeur. Les objets passés en paramètres de la collecte sont des informations du contexte d'exécution requises par l'utilisateur-rice, et qui dépendent du type de nœud sur lequel le metalink est installé. Le paramétrage du metalink de collecte est décrit par le code de la Figure 5.5.

```
1  "collectMetaLink et collectBehavior sont predefinis lors de l'
       initialisation du collecteur"
2  collectMetaLink := MetaLink new.
3  collectMetaLink metaObject: collectBehavior.
4  collectMetaLink selector: #collect:.
5  collectMetaLink control: #after.
6  collectMetaLink arguments: self computeArguments.
7
8  self collectPoints do:[:collectPoint |
9      collectPoint targetAST link: collectMetaLink ]
```

FIGURE 5.5 – Metalink de collecte, avec comme méta-objet l'instance de *CollectBehavior* du collecteur (ligne 2). Ce metalink est installé sur l'AST cible de chaque point de collecte (ligne 7). Après chaque évaluation de l'AST (*#after*, ligne 4), le metalink envoie le message *collect*: (ligne 3) au méta-objet (*collectBehavior*) (ligne 2) avec comme paramètre les éventuels arguments calculés par le collecteur (ligne 5).

CollectPoint. C'est une réification du point de collecte, qui fait référence à une expression précise dans le flot de contrôle. Cette expression est modélisée par un nœud de l'AST (*RBProgramNode*). Un collecteur peut également viser une variable d'instance, une variable de classe ou une variable temporaire. Dans ce cas, un point de collecte est instancié pour chaque accès en lecture et en écriture à la variable. Chaque point de collecte référence un unique metalink de collecte, qui déclenche le comportement de collecte. Par abus de langage, on désignera dans le reste de cette thèse un point de collecte comme étant soit le point de collecte principal ciblant un nœud de l'AST, soit un de ses nœud fils dans le cas de la collecte en profondeur. La différence sera précisée dans le cas où elle présente une importance particulière.

CollectBehavior. Il s'agit du comportement de collecte (prédéfini par chaque collecteur), c'est-à-dire la manière dont un collecteur effectue une collecte lorsqu'un point de collecte est atteint. Ce comportement contrôle donc la collecte des objets et des méta-données afférentes au contexte pour une exécution qui atteint un point de collecte. Pour un collecteur donné, l'unique instance de *CollectBehavior* qui est associée à ce dernier joue le rôle du méta-objet appelé par le metalink de collecte. Lorsque le point de collecte est atteint, le metalink envoie le message *collect*: à l'instance de *CollectBehavior* du collecteur (Figure 5.6).

La méthode *collect*: de la classe *CollectBehavior* prend en paramètre un tableau contenant les données récupérées du point de collecte, c'est à dire le collecteur, l'objet issu de l'évaluation de l'expression ciblée par le point de collecte, et les réifications *primitives* fournies par Reflectivity (ligne 1). Si le collecteur est configuré en mode *replay* pour le nœud de l'AST évalué (voir *OmniscientRecord* ci après), la collecte n'est pas effectuée (ligne 6). Les méta-données issues de transformations spécifiées par l'utilisateur-rice sont alors calculées (ligne 8) puis la garde du collecteur est évaluée (ligne 9). Si la garde n'est pas évaluée au littéral *true*, la collecte n'est pas effectuée. La référence de l'objet est alors collectée (ligne 12). Enfin, si le collecteur est configuré avec une stratégie de gestion mémoire de conservation des objets ou de conservation par copie, une collecte spécifique est effectuée (lignes 14 à 18).

```
1  collect: collectPointData
2      | collector object |
3      collector := collectPointData first.
4      object := collectPointData second.
5
6      collector isInReplayMode ifTrue: [ ^ self ].
7
8      collector applyTransformationsFrom: collectPointData.
9      (collector mustCollect: collectPointData)
10         ifFalse: [^ self ].
11
12     collector basicCollectObject: object.
13
14     collector isOmniscient ifFalse: [ ^ self ].
15
16     self omniscientCollect: object
17          in: collector
18          withReifications: collectPointData
```

FIGURE 5.6 – Comportement de collecte implémenté par la classe *CollectBehavior*.

OmniscientRecord. Chaque fois qu'une collecte est déclenchée, un objet de la classe *OmniscientRecord* est instancié. Un *OmniscientRecord* référence l'objet collecté avec éventuellement une copie de cet objet − en fonction de la stratégie de gestion mémoire paramétrée. L'instance d'*OmniscientRecord* peut être composée d'autres *OmniscientRecord* correspondant à la collecte d'objets *en profondeur* sur l'AST ciblé par le point de collecte. Ces instances sont associées à l'*OmniscientRecord* principal contenant l'objet issu du point de collecte. Un *OmniscientRecord* contient également des méta-données relatives au contexte d'exécution du point de collecte à l'origine de son instanciation.

Les *OmniscientRecord* sont accumulés par un collecteur, à partir de chaque passage par les points de collecte de ce collecteur. Ils annotent le point de collecte à partir duquel ils ont été générés, notamment avec un horodatage. Cela permet au collecteur de reconstruire un historique des évaluations des expressions ciblées par un point de collecte. Si la stratégie de gestion mémoire spécifie la conservation des objets, alors toutes les évaluations sont conservées dans l'historique. Dans le cas contraire, seuls les *OmniscientRecord* dont les objets collectés n'ont pas été réclamés par le gestionnaire de mémoire du programme sont conservés. Lorsque la stratégie spécifie une conservation par copie, les objets sont copiés avec leur état courant au moment de la collecte, et ces copies sont stockées dans la mémoire du programme. Dans Pharo, cette mémoire est partagée entre le programme et les outils présents dans l'environnement d'exécution.

Un *OmniscientRecord* particulier peut être sélectionné par son collecteur pour jouer le rôle de *replay*. Si tel est le cas, l'objet collecté (ou sa copie si elle existe) sera utilisé pour remplacer l'évaluation du noeud de l'AST dont il est originellement issu. Ce *replay* étant une instance de *OmniscientRecord*, il peut être sélectionné parmi le noeud ciblé par le point de collecte et l'un de ses fils − s'il y a eu collecte en profondeur. Ce mécanisme permet de rejouer l'exécution de tout ou partie d'un point de collecte afin de reproduire un comportement observé du programme. Le *replay* est mis en œuvre dynamiquement par l'installation d'un metalink qui remplace l'exécution du nœud correspondant par un renvoi de l'objet collecté ou de sa copie, contenus par l'*OmniscientRecord* sélectionné.

MetaDataRequest. Ces requêtes sont spécifiées par un-e utilisateur-rice ou un outil. Au moment de la collecte, ces requêtes sont évaluées par le *CollectBehavior* du collecteur et produisent des méta-données.

MetaData. Une méta-donnée est une information extraite du contexte d'exécution de l'expression ciblée par un point de collecte. Cette information peut être soit une réification directe du contexte, par exemple le receveur d'un message, soit le résultat d'une transformation à partir d'objets de ce contexte, par exemple le nom

de la méthode active ou la pile d'exécution. Une instance de *MetaData* contenant des transformations effectuées à partir d'objets du contexte du point de collecte contient aussi une copie de ces objets.

Conditions. Une condition est la réification d'une garde de collecte. Il s'agit d'une expression qui doit être évaluée au littéral *true* pour que la collecte soit effective. Plusieurs conditions indépendantes peuvent être définies pour un collecteur. La garde est considérée comme évaluée à *true* si l'évaluation indépendante de chaque condition évalue à *true*. Une condition est exprimée sous forme de bloc, qui prend en paramètre un ensemble de méta-données (instances de *MetaData*) qui peut être vide.

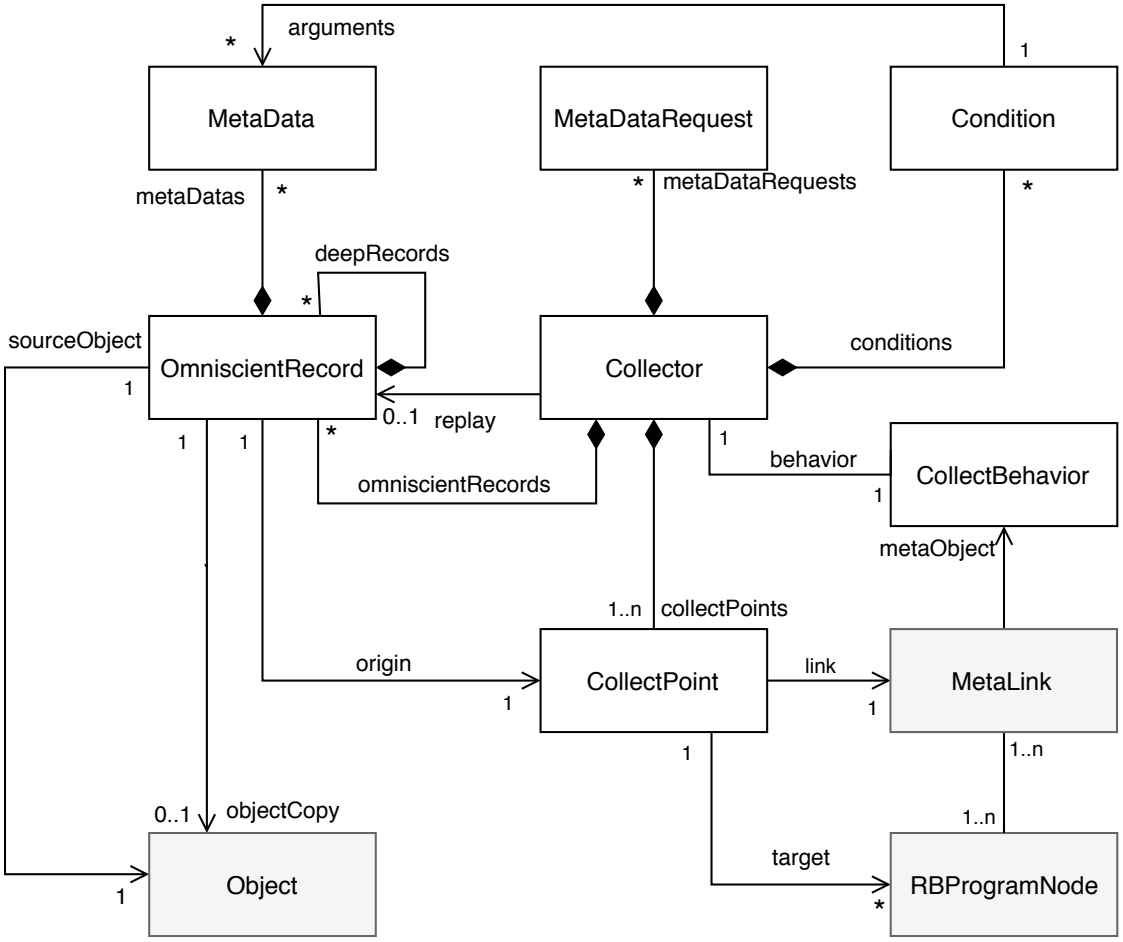


FIGURE 5.7 – Modèle des collecteurs.

5.2.2 Limites du modèle

Le modèle mis en œuvre dans Pharo possède des aspects limitants, notamment concernant les performances du modèle et la consommation mémoire des mécanismes de conservation et de copie des objets collectés. Ces aspects n'ont pas été évalués, et constituent des risques pour le bon fonctionnement du programme en cours d'exécution :

- La collecte en profondeur sur une expression est d'autant plus lente que l'expression est complexe, car un metalink de collecte est installé sur chaque sous-nœud de l'AST du point de collecte.
- Les temps d'installation d'un collecteur, notamment pour la collecte en profondeur, ne sont pas évalués. La mise en place d'un collecteur sur une expression complexe peut donc ralentir le programme dans des proportions non précisées.
- Les stratégies de conservation des objets et de conservation par copie peuvent, pour une seule expression, produire un surplus d'utilisation de la mémoire qui peut saturer la mémoire du programme.

Cependant cette absence de contrainte est importante pour ne pas se limiter dans l'exploration de techniques et d'outils pour le déverminage. C'est le cas en particulier pour investiguer et expérimenter quand les erreurs sont particulièrement difficiles à comprendre et à reproduire. Au-delà de la flexibilité du modèle pour le déverminage, une évaluation est nécessaire pour l'étude et la mise en œuvre de contraintes et d'optimisations permettant au système de passer à l'échelle. Ces problèmes constituent un problème connu des solutions visant à retenir tout ou partie d'un programme [Lewis, 2003]. La littérature fait état de recherches sur les sujets de la sauvegarde d'objets ou du passage à l'échelle de systèmes enregistrant les objets produits par un programme [Lienhard et al., 2006, Pothier et al., 2007, Lienhard et al., 2008, Lienhard et al., 2009, Pothier and Tanter, 2009, Pluquet et al., 2009, Pothier and Tanter, 2011, Infante and Bergel, 2017]. Ces recherches constitueraient le point de départ de l'étude de ces problématiques pour les Collecteurs.

En outre, l'utilisation de Reflectivity comme mécanisme d'instrumentation implique certaines limitations supplémentaires, comme par exemple la restriction aux applications mono-processus ou l'impossibilité d'instrumenter des méthodes présentes sur la pile d'exécution (voir section 4.2.1.2).

5.2.3 L'API des Collecteurs

Cette section décrit l'interface publique des Collecteurs. L'interface peut être utilisée programmatiquement par un-e développeur-euse, ou bien par un outillage.

La classe *Collector* fournit des méthodes globales pour définir des collecteurs, tandis que ses instances permettent d'interagir avec le processus de collecte. L'API peut être utilisée avant et pendant l'exécution d'un programme. La définition et la modification des Collecteurs est dynamique, du moment que le langage hôte le supporte − ce qui est le cas avec notre implémentation sous Pharo. Dans les méthodes d'interface présentées ci-dessous, les mots précédés par le caractère *'#'* sont des paramètres passés à la méthode. Le mot-clé *class* signifie que la méthode présentée est une méthode de classe. Lors de la mise en œuvre, l'API peut être agrémentée de facilités syntaxiques qui ne sont pas présentées ici pour des raisons de simplicité.

5.2.3.1 Interfaces de définition des Collecteurs

Pour définir un collecteur, il faut d'abord définir un point de collecte. Les points de collecte sont réifiés sous quatre formes différentes, pour permettre à l'utilisateur-rice de manipuler des concepts familiers au processus de développement. Les types de points de collecte réifiés dans notre mise en œuvre sont donc les variables temporaires, les variables d'instance, les receveurs d'un message ou les expressions du code source. L'appel d'une méthode de l'interface de définition d'un point de collecte retourne un collecteur stocké dans une collection globale de la classe *Collector* et il est possible d'y accéder à tout moment.

Collecte d'une variable temporaire. Un point de collecte est défini pour chaque lecture ou écriture dans la variable nommée *#tempName*, dans le corps d'une méthode nommée *#methodName* de la classe nommée *#className*.

```
1 Collector class>>
2     collectTemporary: #tempName
3     in: #methodName
4     fromClass: #className.
```

Collecte d'une variable d'instance. Un point de collecte est défini pour chaque lecture ou écriture dans la variable nommée *#ivarName*, dans le corps de toutes les méthodes de la classe nommée *#className* qui référencent la variable − y compris dans ses super classes.

```
1 Collector class>>
2     collectInstVar: #ivarName
3     fromClass: #className.
```

Collecte du receveur d'un message. Un point de collecte est défini pour chaque receveur d'un message nommé *#messageName*, dans le corps d'une méthode nommée *#methodName* de la classe nommée *#className*. Tous les objets qui reçoivent

le message sont collectés, quelle que soit leur classe. Une seconde interface permet à l'utilisateur-rice ou à un outil de préciser la position du receveur dans la méthode, afin de placer le point de collecte sur un seul message. Seul le receveur de ce message dans le corps de la méthode sera alors collecté. Cette position est un entier compris entre un et le nombre de fois où le message est présent dans le corps de la méthode.

```
1  Collector class>>
2      collectReceiversOf: #messageName
3      in: #methodName
4      fromClass: #className.
5
6  Collector class>>
7      collectReceiversOf: #messageName
8      at: position
9      in: #methodName
10     fromClass: #className.
```

Collecte d'une expression. Un point de collecte est défini pour une expression précise dans le flot de contrôle. Cette interface prend directement en paramètre un nœud de la syntaxe abstraite. Cette dernière est peu pratique à manipuler car elle nécessite de naviguer dans l'arbre de syntaxe abstraite d'un programme pour y trouver le nœud représentant l'expression ciblée. Il s'agit d'une méthode destinée à être utilisée par un outil.

```
1  Collector class>>
2      collectReceiverOfNode: #anAstNode.
```

Accès aux collecteurs du système. Tous les Collecteurs présents dans le système peuvent être retrouvés par le biais d'une interface globale (lignes 1-2). Un collecteur particulier peut être retrouvé par son nom (lignes 4-5).

```
1  Collector class>>
2      allCollectors.
3
4  Collector class>>
5      named: aString.
```

5.2.3.2 Interfaces de manipulation des Collecteurs

Les méthodes d'interface suivantes s'appliquent aux instances de la classe *Collector*, et permettent d'interagir avec les Collecteurs. Pour chaque méthode de configuration, par exemple l'ajout d'une condition, il existe une méthode qui annule

cette configuration, par exemple la suppression d'une condition. Pour des raisons de simplicité, ces méthodes ne sont pas présentées ici.

Nommage d'un collecteur. Par défaut un collecteur est instancié par un nom unique, généré par sa méthode d'instanciation. Ce nom peut être modifié s'il reste unique, deux collecteurs ne pouvant porter le même nom.

```
1 Collector class>>
2     name: aString
```

Désinstallation d'un collecteur. La désinstallation d'un collecteur provoque la désinstallation des points de collecte et la relâche de tous les objets collectés, ainsi que de leurs copies éventuelles. Ces objets sont par la suite réclamés par le gestionnaire de mémoire du programme.

```
1 Collector>>
2     release
```

Collecteurs centrés objet. Un collecteur peut être restreint au flot de contrôle d'un objet *anObject* précis, si une référence à cet objet peut être fournie en paramètre. Dans ce cas, la collecte n'est active que pour cet objet précis. L'interface peut être appelée plusieurs fois avec plusieurs objets différents en paramètres. Tous ces objets définissent alors le périmètre de collecte du collecteur.

```
1 Collector>>
2     scopeToObject: anObject.
```

Requêtes de méta-données. Une requête de méta-donnée prend deux formes. Elle peut être une requête de méta-donnée dite *primitive*, c'est-à-dire une réification native de Reflectivity. La méthode *requestReification*: prend en paramètre un symbole correspondant à la réification primitive demandée. La requête peut également être une transformation utilisateur, via la méthode *requestTransformation*:*named*:. Cette méthode prend en paramètre un bloc de code et un nom sous forme de chaîne de caractères. Le bloc peut prendre en paramètre une ou plusieurs réifications primitives, si elle a préalablement été demandée.

Ces deux types de requête produisent, lors de la collecte, des méta-données accessibles par leur nom.

```
1 Collector>>
2     requestReification: aSymbol.
3
4 Collector>>
5     requestTransformation: aBlock named: aString.
```

Traces. La trace est une transformation particulière, stockée sous forme de chaîne de caractères. Elle prend la forme d'un bloc de code, qui peut prendre en paramètre des méta-données. Les méta-données sont identifiées par leurs noms, dans les arguments du bloc de code.

```
1 Collector>>
2     trace: aBlock.
```

Garde de collecte. La garde est composée de conditions indépendantes. Chaque condition est spécifiée sous la forme d'un bloc de code qui peut prendre en paramètre des méta-données, identifiées par leurs noms, dans les arguments du bloc.

```
1 Collector>>
2     addCondition: aBlock.
```

Collecte en profondeur. Un collecteur particulier peut être configuré pour effectuer une collecte en profondeur. La méthode *deepCollect*: prend en paramètre un booléen, qui, en fonction de sa valeur, active ou désactive la collecte en profondeur sur tous les points de collecte définis par le collecteur.

```
1 Collector>>
2     deepCollect: aBoolean.
```

Sélection des stratégies de gestion mémoire. Trois types de gestion mémoire peuvent être sélectionnés par un symbole correspondant à une stratégie particulière :

- *#weak* : conservation par référence faible ;
- *#strong* : conservation par référence forte ;
- *#omniscient* : conservation par copie.

Dans le cas de la conservation par copie, l'évolution complète de l'état d'un objet est conservée en mémoire et produit un historique de l'évaluation

```
1 Collector>>
2     selectObjectStorageStrategy: aSymbol.
```

5.2.4 Mise en œuvre avec Python et Reflectivity

En tant que patron de langage, les Collecteurs peuvent être mis en œuvre dans d'autres langages dynamiques, comme par exemple Python [Python, 2017]. Cependant, la mise en œuvre décrite dans ce chapitre est fortement liée à Reflectivity, qui n'existe pas en dehors de Pharo. Nous avons donc implémenté Reflectivity avec

Python, dans une mise en œuvre utilisant l'interpréteur *PyPy*[1]. Cette version de Reflectivity[2] est partielle : nous n'avons porté que le cœur de Reflectivity, contenant les metalinks, les réifications dépendant des nœuds de l'AST ainsi que le contrôle appliqué à l'exécution des metalinks lorsqu'un noeud annoté est atteint dans le flot de contrôle du programme en cours d'exécution.

Nous avons ainsi pu expérimenter une mise en œuvre des Collecteurs avec Python[3]. L'API Python des Collecteurs est similaire à celle de Pharo (section 5.2.3). Elle permet les mêmes fonctionnalités, notamment la spécification des points de collecte dans le flot de contrôle et la collecte des objets qui en sont issus à l'exécution du programme, et de manière complètement non-anticipée. L'API et le modèle sont bien sûr légèrement différents, car les briques de base du langage diffèrent entre Pharo et Python − notamment la syntaxe, la grammaire et l'arbre de syntaxe abstraite. La mise en œuvre Python n'est pas non plus autant développée qu'avec Pharo. En particulier, les outils n'ont pas été entièrement re-développés avec Python. Seul l'outillage de définition et de configuration des Collecteurs (décrits par la section suivante) ont été mis en œuvre avec l'environnement de développement Python *PyCharm*[4]. Les stratégies de gestion mémoire de conservation des objets n'ont pas été mises en œuvre, et les objets sont conservés par références faibles. Il s'agit principalement d'une expérimentation destinée à montrer qu'en tant que patron de langage, des mises en œuvre dans d'autres langages dynamiques sont possibles.

Le support de l'instrumentation non-anticipée n'est pas non plus le même dans Pharo et Python. Pharo est un environnement de *Live-Programming*, tel que décrit dans l'état de l'art (chapitre 2). Notre mise en œuvre des Collecteurs et leurs outils avec Python repose sur une solution dédiée liée à l'environnement *PyCharm*, sous forme de *plug-in*. D'autres supports décrits dans l'état de l'art pourraient servir de base à cette mise en œuvre Python, comme par exemple des solutions de mise à jour à chaud pour Python [Martinez et al., 2013, Martinez et al., 2015].

5.3 Outils pour l'application au déverminage

Plusieurs outils ont été mis en œuvre dans Pharo pour expérimenter l'application des Collecteurs au déverminage de programmes à objets. Cette section présente tout d'abord l'outillage mis en place pour la définition et la configuration des Collecteurs. Puis, nous présentons l'inspecteur de collecteurs, qui permet d'explorer l'historique d'un point de collecte et de remonter dans le temps dans les diverses exécutions de l'expression instrumentée. Ces outils reposent sur le modèle et l'API présentés par la section 5.2.

1. https ://pypy.org/
2. https ://git.kloum.io/StevenCostiou/Reflectivity-PyPy
3. https ://git.kloum.io/StevenCostiou/Py-Collectors
4. https ://www.jetbrains.com/pycharm/

5.3.1 Outillage des Collecteurs

Plusieurs outils permettent de spécifier puis d'interagir avec les Collecteurs. Tous ces outils permettent une interaction dynamique avec le code source représentant des objets (par exemple, une variable). La collecte d'objet peut être spécifiée avant ou pendant l'exécution du programme, directement dans l'environnement d'exécution du programme.

Définition des points de collecte. Les points de collecte peuvent être définis à partir d'un menu contextuel dans le code source de Pharo (Figure 5.8). L'utilisateur-rice doit sélectionner une expression cible et choisir le type de point de collecte dans le menu – respectivement une variable d'instance, une variable temporaire, le receveur d'un message ou une expression générique. La Figure 5.9 illustre la visualisation du code source après la définition d'un collecteur. Le point de collecte est surligné en orange, et un bouton dans la gouttière permet d'accéder à la configuration, à la visualisation et à l'option de désinstallation du collecteur.

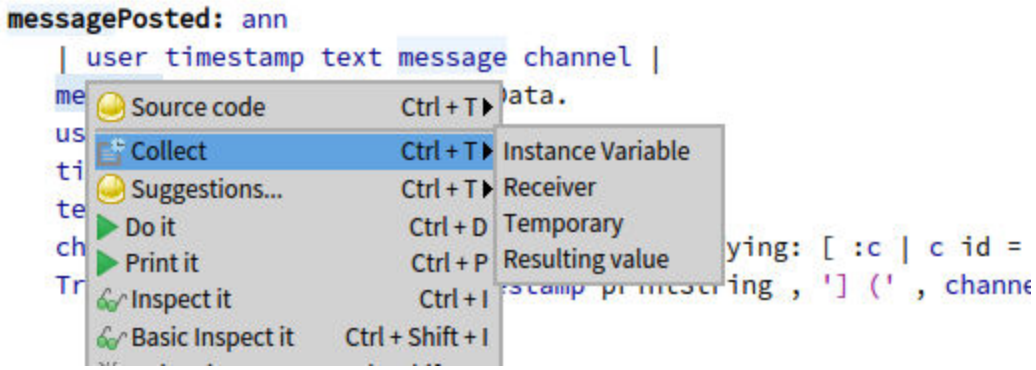


FIGURE 5.8 – Menu de définition de collecteurs à partir du code source.

Configuration d'un collecteur. Chaque collecteur peut être configuré (Figure 5.10). La configuration d'un collecteur s'applique à tous ses points de collecte.

1. Visualisation du ou des points de collecte dans le code source.
2. Réifications *primitives* (Reflectivity) demandées parmi celles disponibles pour le type de point de collecte mis en place par le collecteur (variables, receveur d'un message, expression). Dans cet exemple, la réification *value* est demandée, et correspond à l'objet issu de l'évaluation du point de collecte. Le choix de ces réifications est illustré par la Figure 5.11.

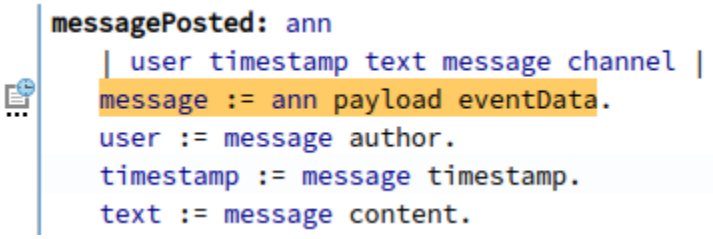


FIGURE 5.9 – Visualisation des points de collecte dans le code source.

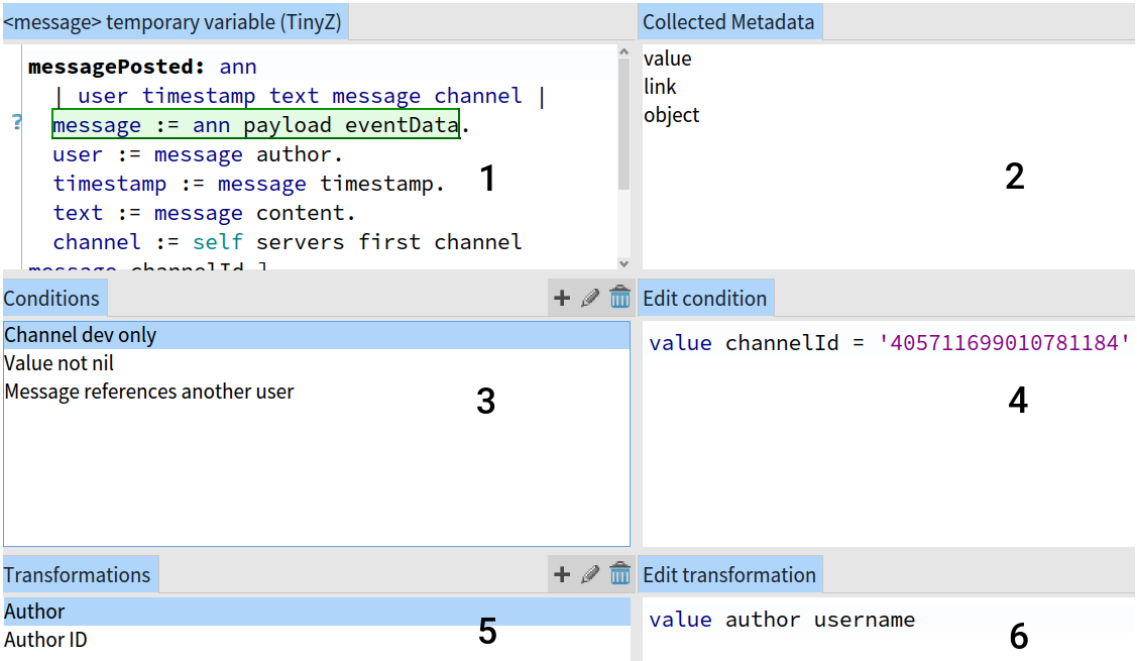


FIGURE 5.10 – Configuration d'un collecteur. 1) point de collecte, 2) réifications (requêtes de méta-données), 3) et 4) conditions, 5) et 6) transformations (requêtes de méta-données utilisateur).

3. Liste de conditions, nommées par l'utilisateur-rice. Leur évaluation indépendante constitue la garde de collecte, et toutes les conditions doivent être évaluées au littéral *true* pour que la collecte soit effectuée.
4. Spécification d'une condition. Le résultat de l'évaluation de la condition doit être un booléen. Dans le cas contraire (ou d'une erreur), le littéral $false$ est renvoyé par défaut. Le corps de la condition peut référencer les réifications *primitives* (par exemple ici *value*) ou une transformation nommée (point 5).

5. Liste de transformations nommées (point 6).
6. Expression d'une transformation utilisateur. Le corps de la transformation peut directement référencer les réifications *primitives* (ici *value*).

Les conditions et les transformations peuvent être modifiées dynamiquement sans réinstallation du collecteur. Les modifications sont alors prises en compte à la prochaine collecte, pour les prochains objets collectés. Les objets déjà collectés ne sont donc pas soumis à ces modifications. La modification des réifications *primitives* fournies par Reflectivity nécessite une réinstallation du collecteur (Figure 5.11).

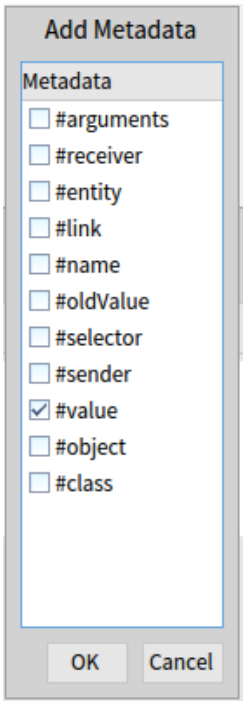


FIGURE 5.11 – Sélection des réifications *primitives* fournies par Reflectivity. La liste de réifications disponibles peut varier selon le type de nœud ciblé par les points de collecte.

Collecteurs centrés objet. Un collecteur peut être restreint au flot de contrôle d'un seul objet. Cela signifie que la collecte ne s'effectue que lorsque cet objet précis rencontre et évalue un point de collecte. Un nouvel onglet est intégré dans l'inspecteur d'objet du système (Figure 5.12). Tous les collecteurs présents dans le système sont visibles, et un menu contextuel permet de restreindre le périmètre du collecteur sélectionné à l'objet inspecté. Le périmètre centré objet d'un collecteur peut comprendre plusieurs objets différents.

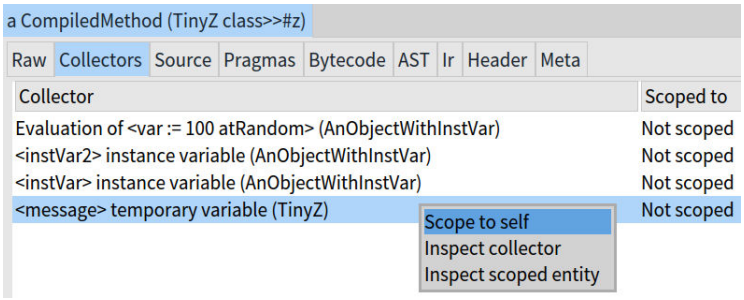


FIGURE 5.12 – Inspecteur natif de Pharo ouvert arbitrairement sur un objet du système : un nouvel onglet est intégré pour visualiser les collecteurs et restreindre leur périmètre de collecte à un ou plusieurs objets spécifiques. Ici la commande "*Scope to self*" ajoutera l'objet inspecté au périmètre de collecte du collecteur sélectionné.

5.3.2 Visualisation des objets collectés : les historiques de collecte

Un inspecteur dédié permet de visualiser, pour chaque collecteur, l'historique des collectes pour tous les points de collecte du collecteur inspecté. L'historique d'un point de collecte particulier est également accessible à partir de sa visualisation propre dans le code source de la méthode instrumentée (Figure 5.9). Un historique pour un point de collecte particulier est illustré par la Figure 5.13, et détaillé par la Figure 5.14. La précision d'une visualisation dépend de la stratégie de gestion mémoire. Les objets conservés par référence peuvent évoluer au cours de l'exécution et s'ils ont été collectés plusieurs fois, leur état est le même quelque soit le point sélectionné dans l'historique. La conservation par copie permet de tracer l'état d'un objet au travers des différentes collectes qui le concernent. Le détail de la visualisation d'un historique, présenté par la Figure 5.14, contient les éléments suivants :

1. Chaque collecte est affichée dans un historique ("*Collection time*"). La mention "*value change*" précise si, entre deux collectes, l'objet collecté a changé. Lorsqu'un élément de l'historique est sélectionné, et dans le cas d'une collecte en profondeur, la liste des sous-expressions constituant le point de collecte est affichée ("*Collected sub-expressions*").
2. Lorsqu'une expression ou une sous-expression est sélectionnée, différents éléments contextuels deviennent disponibles à la visualisation :
   a) La visualisation du code source de l'expression pour laquelle le résultat de l'exécution a été collecté. Chaque expression sélectionnée sera mise en avant dans le code (avec un fond vert) dans l'expression globale ciblée par le point de collecte (avec un fond orange).

b) Pour chaque expression sélectionnée, l'objet collecté – ou sa copie selon la stratégie de gestion mémoire sélectionnée – est ouvert dans un inspecteur dédié. Cet inspecteur présente l'objet en lui-même, et les méta-données collectées.

3. Pour chaque collecte (point 1), la pile d'exécution qui a mené à la collecte est présentée. La comparaison des piles d'exécution de deux collectes différentes permet alors de comparer les objets collectés au regard du *chemin* emprunté par le programme pour arriver à leur collecte. Cette pile contient une liste des dernières méthodes exécutées jusqu'à la collecte, avec leur code source et leur contexte. Il est alors possible de visualiser l'état des variables et des objets au travers de la pile, jusqu'à la collecte. La taille de la pile est limitée par un paramètre global et peut être augmentée, mais il faut alors considérer l'impact qu'aura sur la mémoire la sauvegarde de trop grandes piles d'exécution. La limite est fixée de manière arbitraire à 20 méthodes dans cette mise en œuvre.
4. S'il y a eu collecte en profondeur, une sous-expression du point de collecte peut jouer le rôle de *replay*. À l'exécution, l'objet collecté pour le nœud sélectionné remplacera la valeur de retour de l'évaluation de la sous-expression correspondante. Cette dernière n'est donc pas évaluée. L'activation du *replay* dans la Figure 5.14 remplacera l'évaluation de l'expression sélectionnée (2a, en vert) par un retour de l'objet collecté (2b). La mise en place d'un *replay* désactive la collecte pendant sa durée d'activation. L'activation et la désactivation du *replay* peuvent être effectuées avant ou pendant l'exécution du programme.

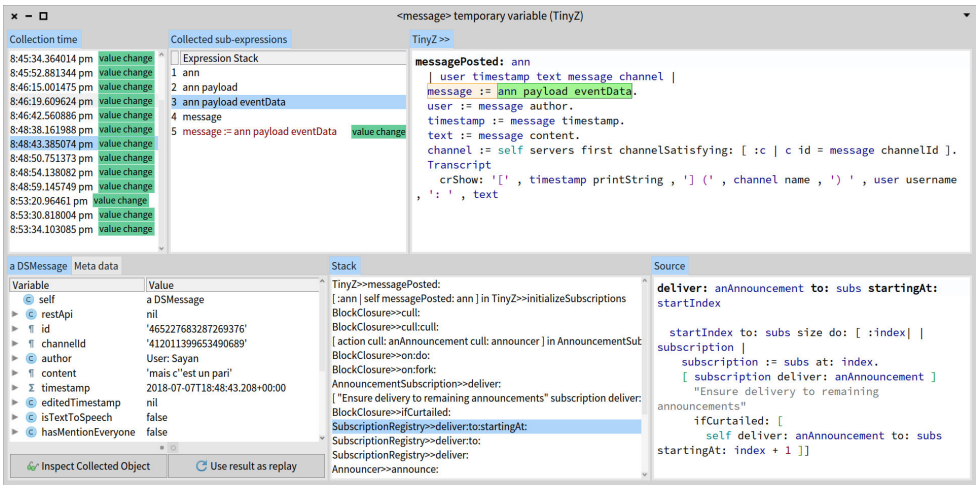


FIGURE 5.13 – Vue d'ensemble d'un inspecteur ouvert sur un point de collecte. Les différentes parties de l'inspecteur sont détaillées par la Figure 5.14.

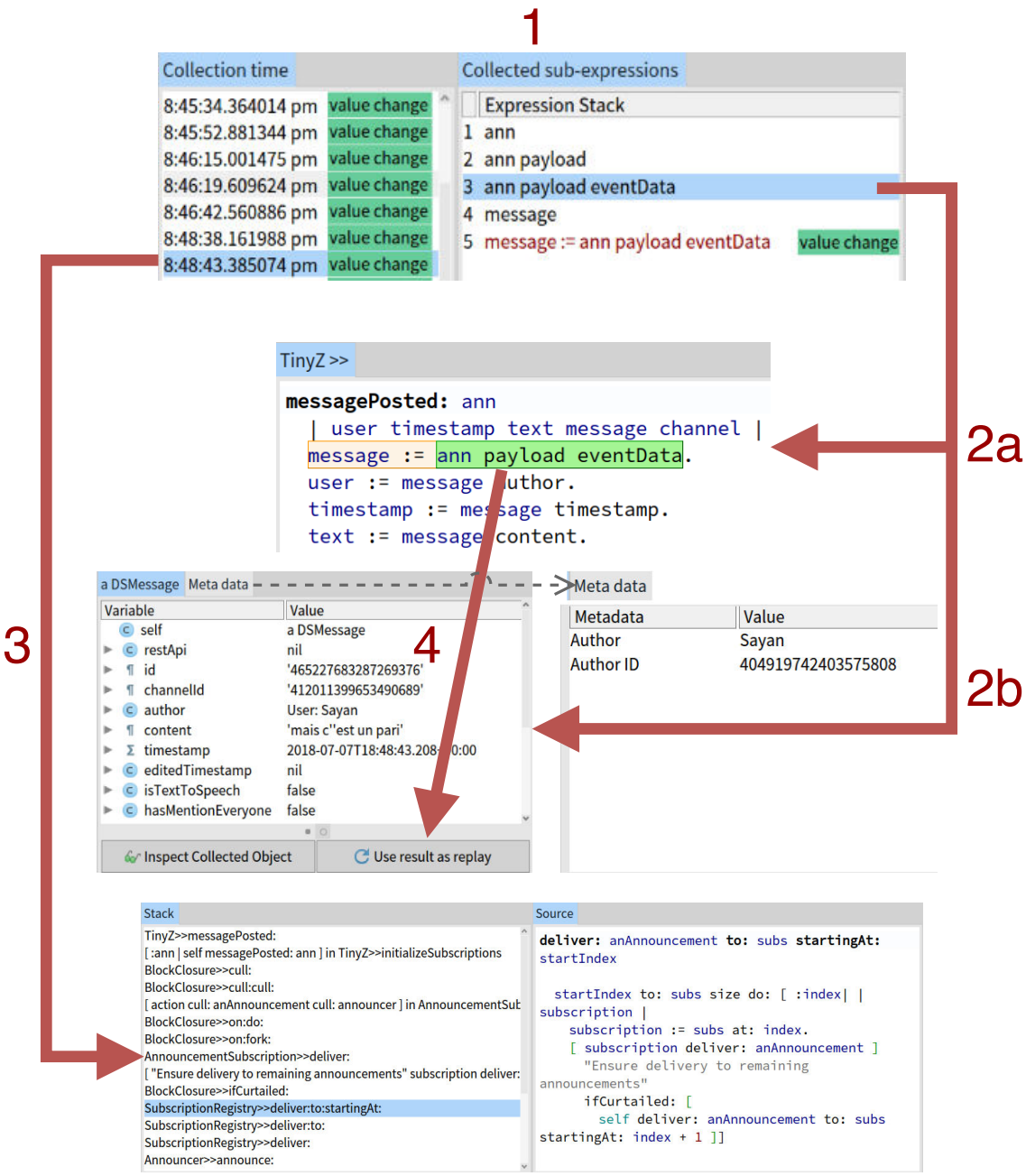


FIGURE 5.14 – Détail d'un historique de point de collecte. 1) Historique et sous-expressions soumises à la collecte. 2a) Visualisation du point de collecte (en orange) et de la sous-expression sélectionnée (en vert). 2b) Objet collecté pour la sous-expression sélectionnée. 3) Pile d'exécution qui a mené à la collecte. 4) Remplacement de l'évaluation d'une expression sélectionnée par son objet collecté pour les prochaines exécutions (*replay*).

5.4 Traces d'objets non-anticipées dans un programme en cours d'exécution

Des traces sont des informations issues de l'exécution d'un programme. La technique, dite du *printf*, est un des premiers moyens auxquels un-e développeur-euse a recours pour observer le comportement et l'état du programme (voir section 1.2.1.2). Dans cette section, nous illustrons un cas d'utilisation sur une application en cours d'exécution que nous souhaitons observer par le biais de traces d'exécution. L'application est un *chat-bot*, un robot connecté à un serveur de discussion qui fournit un certain nombre de services. Le *bot* reçoit des messages des utilisateur-rice-s qui souhaitent accéder à un service, interprète ces messages et fournit le service demandé. Nous utilisons les collecteurs et leur outillage pour spécifier, collecter et tracer des objets pendant l'exécution du programme.

5.4.1 Description du problème

L'application du *chat-bot* est en cours d'exécution, et les services qu'il fournit ne doivent pas être interrompus. Nous souhaitons produire des traces des messages reçus à partir du serveur de discussion pour observer le comportement du programme. Mais cette fonctionnalité n'est pas présente dans le code du programme et nous ne souhaitons pas introduire d'instrumentations dans le code. En particulier, les contraintes suivantes s'appliquent :

Sélection des objets tracés. Tous les messages reçus ne nous intéressent pas, et nous souhaitons conditionner l'instrumentation. Seuls les messages satisfaisant certaines conditions précises doivent être tracés.

Modification dynamique des instrumentations. Lors de l'exploration du comportement du programme, et potentiellement d'un *bug*, la modification des instrumentations permet d'affiner et de filtrer les informations tracées. Il est nécessaire de pouvoir modifier facilement et régulièrement les éléments tracés et les conditions sous lesquelles les traces sont actives.

Instrumentation sans interruption de service. Pour l'utilisateur-rice l'insertion des traces doit être transparente, et ne doit pas impliquer d'interruption de service. Les deux contraintes précédentes doivent pouvoir s'appliquer alors que le programme est en cours d'exécution.

Séparation des préoccupations. Le code de la méthode de réception de message par le *chat-bot* est illustré par la Figure 5.9. L'insertion directe d'instrumenta-

tions dans cette méthode est problématique, car le code sera immédiatement pollué par ces dernières. Les instrumentations ne doivent pas modifier le code de base du programme, et doivent être indépendantes de ce dernier.

5.4.2 Traces dynamiques d'objets collectés

Nous avons expérimenté l'insertion dynamique de traces à partir d'objets collectés dans l'application du *chat-bot*. Cette expérimentation est illustrée dans les différentes figures de ce chapitre. La Figure 5.8 illustre la définition d'un collecteur sur une variable temporaire nommée *message*, qui stocke un message entrant. La Figure 5.9 montre une visualisation du code représentant le message qui sera collecté. La Figure 5.10 montre la configuration finale du collecteur. Trois méta-données sont demandées (2), la valeur de l'objet (la référence du message), le méta-lien à l'origine de la collecte (c'est à dire l'instrumentation sous forme de *metalink*) ainsi que l'objet qui exécute la méthode dans laquelle est collecté le message. Trois conditions de collecte sont spécifiées (3) : les messages collectés sont uniquement les messages avec un texte non vide, émis dans un canal de communication particulier, et dont le texte référence un autre utilisateur-rice. Une liste de transformations (5) spécifie le comportement de trace. Le nom de l'auteur du message et son identifiant sont tracés chaque fois qu'un message satisfaisant ces conditions est reçu par le *chat-bot*. Les objets collectés sont visualisables dans l'historique de collecte des Figures 5.13 et 5.14. Un exemple de message collecté et des traces générées sont visibles dans le détail de la Figure 5.14 (partie 2b). La partie gauche montre un inspecteur sur l'objet collecté, tandis que la partie droite montre le résultat des traces.

5.4.3 Discussion

La condition de collecte permet de sélectionner les objets à collecter. Comme une trace est générée pour tous les objets collectés, cela permet de conditionner la trace. À tout moment, l'utilisateur-rice peut rajouter une nouvelle condition de collecte, ou modifier des conditions existantes. L'ajout et la modification de conditions sont alors pris en compte lors de la collecte suivante. Cela s'applique également aux transformations utilisées pour générer des traces. Il est donc possible de dynamiquement faire évoluer les traces générées, suivant le besoin des développeur-euse-s, alors que le programme est toujours en cours d'exécution. Enfin, l'instrumentation de collecte étant non-intrusive − le code de la méthode avec le collecteur n'est pas modifié − la préoccupation de tracer certains objets précis du programme est totalement cloisonnée et séparée du code de base du programme.

La méthode introduite par l'outillage des collecteurs pour générer des traces est à rapprocher des outils de traces que sont les *Tracepoints* et de la programmation par aspects (AOP), étudiés à la section 1.2.2. Par rapport aux *Tracepoints*, les

collecteurs fournissent la possibilité de réifier toutes les informations contextuelles disponibles du nœud de l'AST sur lequel ils sont installés, ainsi que des transformations de l'objet collecté librement spécifiées par l'utilisateur-rice. L'AOP est surtout orienté vers une séparation stricte des préoccupations, et ne fournit pas, à notre connaissance, de modèles ni d'outillage permettant de sélectionner de telle manière les objets sur lesquels appliquer des traces. L'AOP pourrait en revanche servir de support pour une autre mise en œuvre des collecteurs.

5.5 *Record/Replay* pour la reproduction de valeurs non-déterministes

Le déverminage *Record/Replay* [Ronsse and De Bosschere, 1999, Wang et al., 2014] consiste à enregistrer l'exécution du programme afin de la rejouer. Cela facilite l'observation des bugs qui effacent leurs traces après avoir affecté le système [Eisenstadt, 1997] ou des bugs non-déterministes, difficiles à reproduire [Grottke and Trivedi, 2007, Grottke et al., 2008]. Dans cette section, nous reprenons de la littérature un exemple de bug non-déterministe pour lequel il est intéressant de pouvoir rejouer l'exécution du programme. Nous illustrons l'utilisation de la fonctionnalité *replay* de l'outillage des collecteurs pour enregistrer et reproduire le problème.

5.5.1 Description du problème

Le problème suivant est décrit par Shulz [Schulz, 2017, Schulz and Bockisch, 2017] et illustré par la Figure 5.15. Le problème vient de l'instruction *rnd.nextInt()*, qui génère un entier aléatoire. Cette méthode retourne très exceptionnellement une valeur que n'est pas capable d'interpréter la méthode *Math.abs()*. Cette dernière doit calculer la valeur absolue de l'entier qui lui est passé en paramètre, donc par définition elle doit retourner une valeur positive. Or, lorsque la valeur particulière issue de l'exécution de l'expression *rnd.nextInt()* lui est passée en paramètre, *Math.abs()* retourne une valeur négative.

Comme analysé par Schulz dans son exemple [Schulz and Bockisch, 2017], la source de ce problème n'est pas facile à observer et à comprendre avec des méthodes traditionnelles. Au moment où la valeur est retournée, le résultat aléatoire de *rnd.nextInt()* a été *consommé* par l'appel à *Math.abs()* et le modulo qui suit (*%n*). La valeur intermédiaire n'est donc plus observable. Placer un point d'arrêt avant la méthode ne permet pas de reproduire le problème, car une partie du comportement est non-déterministe et le problème ne se produit pas à chaque exécution de la méthode. L'autre possibilité est de modifier manuellement le code du programme, pour y introduire des variables temporaires qui capturent la génération

du nombre aléatoire, ainsi que du code conditionnel pour ouvrir une visualisation (dévermineur, traces...) uniquement lorsque le résultat est problématique.

Listing 1: Number generator example [1].

```
static Random rnd = new Random();
static int randomPositive(int n) {
  return Math.abs(rnd.nextInt()) % n;
}
```

FIGURE 5.15 – Problème non-déterministe, extrait de [Schulz and Bockisch, 2017] : *Math.abs()* retourne parfois un nombre négatif au lieu d'une valeur absolue.

Nous reproduisons ce problème dans Pharo, dans un exemple simplifié pour la démonstration. Il s'agit d'une instance d'une classe *PharoRandom*, qui implémente la méthode *randomPositive*: prenant un paramètre. Ce paramètre est un entier qui définit la borne supérieure du résultat aléatoire. Le code de la méthode *randomPositive*: est le suivant :

```
1 randomPositive: n
2     ^ (self abs: rnd nextInt) modulo: n
```

Le générateur d'entiers est référencé par la variable d'instance *rnd*. Le tirage aléatoire est effectué par l'instruction *rnd nextInt*. Nous mettons en œuvre dans la méthode *abs*:, qui prend en paramètre le résultat aléatoire, une instrumentation permettant de temps à autre et suivant le résultat du tirage de renvoyer un résultat négatif. Nous pouvons alors reproduire artificiellement le problème évoqué à la section précédente : très rarement, le résultat renvoyé par la méthode *randomPositive*: est négatif.

5.5.2 Rejouer une exécution avec un objet collecté

Nous nous intéressons à observer et à déboguer le comportement de la méthode *randomPositive*: lorsque le résultat qu'elle renvoie est négatif. Il nous faut pour cela reproduire le résultat du tirage aléatoire qui provoque cette erreur. Nous définissons un collecteur sur l'expression dont l'exécution produit le résultat final. Nous configurons le collecteur pour n'effectuer la collecte que dans le cas où le résultat de l'exécution de cette expression est négatif. La configuration du collecteur est illustrée par la Figure 5.16. L'expression dont le résultat sera collecté est surlignée en vert par l'éditeur (en haut) et la condition de collecte est exprimée en dessous. De

cette manière, la collecte ne s'effectuera que lorsque le résultat de l'évaluation de cette expression sera un entier négatif.

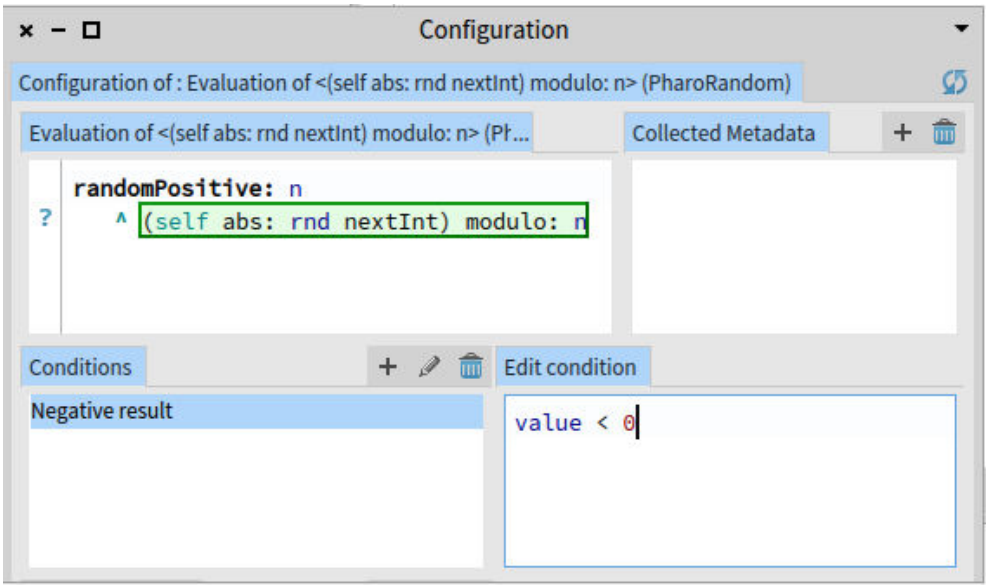


FIGURE 5.16 – Configuration d'une collecte conditionnelle sur l'expression cible (en vert). La collecte ne s'effectue que si le résultat de l'expression est inférieur à zéro.

L'utilisation de la méthode *randomPositive*: finit, à terme, par produire un résultat négatif et une collecte est effectuée. Le résultat de la collecte est visible à la Figure 5.17. La pile d'exécution de l'expression totale est visible à gauche de la fenêtre. Les objets résultant de l'évaluation de chaque sous-expression de l'expression principale ont également été collectés, et peuvent être inspectés individuellement. Lorsqu'une sous-expression est sélectionnée, elle est indiquée par une couleur différente dans la visualisation du code source (à droite) et l'objet collecté pour cette sous-expression est ouvert dans un inspecteur (en bas). Nous pouvons alors observer que l'expression principale renvoie un résultat négatif lorsque la sous-expression *rnd nextInt* renvoie l'entier 19. Il y a donc un problème quelque part dans le reste de l'expression, qui finit par renvoyer un entier négatif lorsque l'entier 19 est tiré aléatoirement.

Nous utilisons alors la fonctionnalité *replay* des collecteurs. Le bouton *use result as replay* visible à la Figure 5.17 permet de configurer la sous-expression sélectionnée pour être systématiquement rejouée avec le résultat collecté visible dans l'inspecteur. La collecte est alors désactivée pour le point de collecte complet, comprenant l'expression principale et toutes ses expressions. Lors de l'exécution de cette méthode, la sous-expression configurée en mode *replay* ne sera plus exécutée. À la place, l'objet collecté configuré pour le *replay* sera directement retourné et utilisé par le reste de la

méthode. La Figure 5.18 montre la méthode *randomPositive*: dans le navigateur de classe de Pharo avec un collecteur en mode *replay* configuré sur la sous-expression *rnd nextInt*. L'outillage surligne l'expression instrumentée et indique dans la marge à gauche qu'il s'agit d'un *replay* (en haut). Lorsque l'utilisateur-rice passe le curseur de la souris sur cette indication, un menu apparaît et permet de visualiser l'objet qui sera retourné au lieu de l'exécution de l'expression surlignée, ainsi que de désinstaller le *replay* pour retrouver un comportement non-instrumenté.

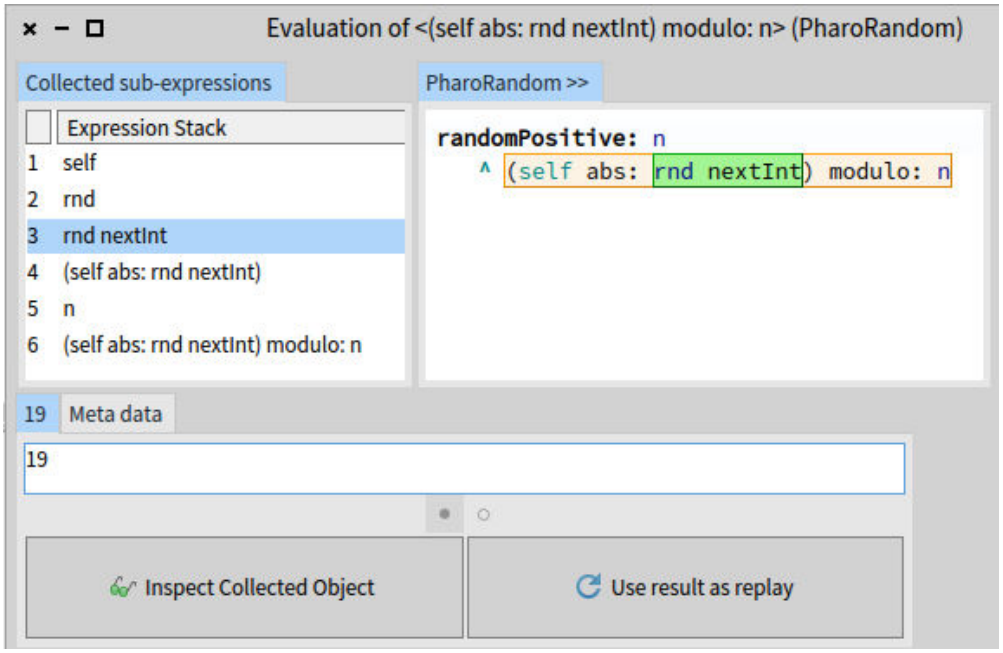


FIGURE 5.17 – Visualisation d'une collecte. La sous-expression *rnd nextInt* (sélectionnée à gauche et surlignée dans le code source à droite) produit un résultat aléatoire (l'entier 19, visible en bas) qui provoque une erreur. Le bouton *use result as replay* permet de remplacer les exécutions futures de l'expression sélectionnée par l'objet collecté (l'entier 19).

Une fois le *replay* configuré avec la valeur problématique, il n'y a plus de composante aléatoire dans le comportement de la méthode. Il est alors possible d'observer pourquoi cette valeur particulière produit un résultat négatif alors que la méthode doit produire un entier positif. Cette observation peut se faire en plaçant un point d'arrêt au début de la méthode. Un déverminer s'ouvrira et l'utilisateur-rice pourra déboguer pas à pas l'exécution de la méthode. L'utilisateur-rice ne pourra pas entrer avec le débogueur dans la méthode *nextInt*, car le résultat de son exécution est directement retourné par le collecteur en mode *replay*. En revanche, la méthode peut être exécutée autant de fois que nécessaire pour comprendre pourquoi la valeur

rejouée (ici l'entier 19) produit un résultat non conforme à celui attendu. Une fois que le problème est compris, corrigé et testé, le collecteur peut être désinstallé.

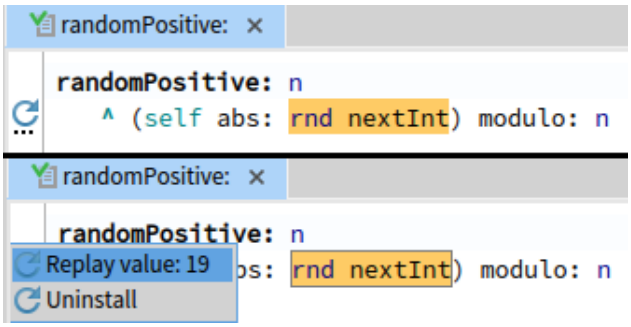


FIGURE 5.18 – Visualisation d'une expression dont l'exécution est remplacée par un objet fixe. En haut, la méthode avec l'indicateur de *replay* dans la gouttière. En bas, le menu contextuel de l'indicateur de la gouttière.

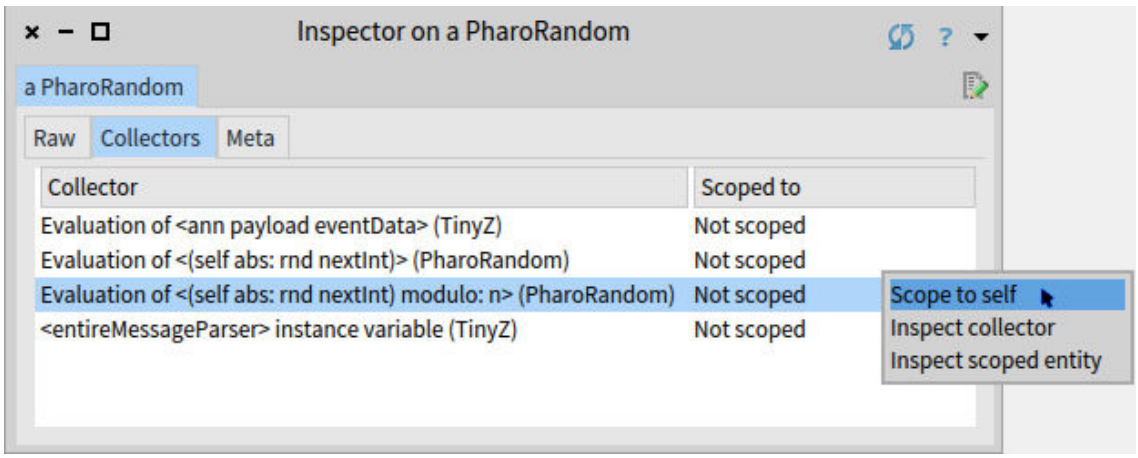


FIGURE 5.19 – Restriction d'une opération de collecte et de *replay* à un objet en utilisant l'inspecteur de Pharo.

Il est possible de restreindre les opérations de collecte et de *replay* à un seul objet en utilisant l'interface *scopeToObject*: des collecteurs et en passant l'objet visé en paramètre (section 5.2.3). Cette interface a été intégrée dans les outils de l'environnement de développement de Pharo pour faciliter son utilisation. La Figure 5.19 montre l'inspecteur de Pharo ouvert sur l'instance de la classe *PharoRandom* utilisée dans la reproduction du *bug* décrit ci-dessus. La liste des collecteurs du système

est visible dans l'onglet *Collectors* de l'inspecteur. Chaque collecteur peut être individuellement restreint à l'objet inspecté, par le biais du menu contextuel. Dans ce cas d'utilisation précis, les opérations de collecte et de *replay* ne seront actives que pour l'objet auquel le collecteur aura été restreint. Un collecteur peut, au travers de cette interface, être restreint à un seul objet ou à un ensemble d'objets spécifiques.

5.5.3 Discussion

L'exemple présenté est une reproduction artificielle d'un problème afin d'illustrer l'utilisation des collecteurs et de leur outillage *Record/Replay* pour l'investigation de *bugs* non-déterministes. L'outillage permet intuitivement de sélectionner les expressions à instrumenter et les parties de ces expressions à rejouer.

Avantages. L'utilisation d'objets collectés et enregistrés pour remplacer localement l'exécution des expressions dont ils sont issus permet d'éliminer leur aspect non-déterministe. Il est alors possible de reproduire un problème, et l'exécution du programme peut être observée pour comprendre pourquoi les valeurs qui sont rejouées provoquent ce problème. La définition de collecteurs peut être non-anticipée. Il est donc possible de définir un collecteur et de spécifier un objet collecté pour jouer le rôle de *replay* pour l'expression dont il est issu alors même que le programme est toujours en cours d'exécution. La possibilité de mettre en place du *replay* dans un programme en cours d'exécution permet d'observer un problème dans son contexte d'apparition tout en contrôlant certains aspects de ce contexte.

L'utilisation de traces conditionnelles et non-intrusives aurait également permis de trouver la valeur problématique, et par exemple de mettre au point des tests unitaires permettant de reproduire et de corriger le problème. Cependant, il faut que les traces soient suffisamment complètes pour pouvoir décrire un objet plus complexe qu'un entier, et que l'outillage puisse reconstruire cet objet pour reproduire le problème dans un test. Comme les collecteurs enregistrent les objets dans la mémoire du programme, la fonctionnalité de *replay* de leur outillage permet de rejouer une exécution locale pour des objets complexes.

Limitations. Rejouer le résultat d'une expression enregistrée ne rejoue pas l'exécution de cette expression. Par exemple si cette expression contient du code dont l'exécution modifie l'état du programme, ce code n'est plus exécuté lors du *replay*. À la place, l'objet enregistré et configuré comme valeur de *replay* est directement retourné comme résultat de l'expression. Cela pose deux problèmes potentiels. D'une part, si la modification de l'état du programme appliqué par le code de l'expression en *replay* n'est plus exécuté, l'état du programme n'est plus cohérent par rapport à une exécution non-instrumentée. Cela peut provoquer de nouveaux problèmes, ou

encore empêcher l'apparition du problème recherché. D'autre part, comme ce code n'est plus exécuté, il n'est pas possible de l'exécuter pas-à-pas pour observer la production de la valeur enregistrée dans le dévermineur. Le *replay* porte donc sur les objets produits par l'exécution, et non pas sur la possibilité de rejouer une exécution de code enregistrée.

Travaux similaires. *Redshell* [Schulz, 2017, Schulz and Bockisch, 2017] est un outil pour le dévermineur Java qui enregistre localement à une méthode les valeurs intermédiaires produites par l'exécution. Lorsque le débogueur s'ouvre, il est alors possible d'observer ces valeurs intermédiaires, ce qui revêt un caractère particulier lorsqu'elles sont non-déterministes. Il n'est cependant pas possible de les rejouer localement, et toutes les expressions de toutes les méthodes sont instrumentées tandis que les collecteurs permettent de cibler et de conditionner manuellement les expressions à enregistrer. *BigDebug* [Gulzar et al., 2016a, Gulzar et al., 2016b, Gulzar et al., 2017] permet de simuler des points d'arrêts dans des applications distantes, et de rejouer localement l'exécution du contexte récupéré du programme distant. À l'instar des collecteurs, il est possible de placer des gardes qui conditionnent la récupération des contextes du programme en cours d'exécution. D'une manière générale, les possibilités décrites dans cette section sont fournies par les dévermineurs *Record/Replay* [Ronsse and De Bosschere, 1999, Wang et al., 2014]. Dans ces débogueurs en revanche, c'est l'exécution complète des parties du programme enregistré qui est rejouée (par exemple une méthode) tandis que les collecteurs remplacent l'exécution d'un bout de code par le renvoi d'un objet enregistré.

5.6 Historiques de collecte pour l'investigation de problèmes difficiles

Cette section décrit une expérimentation des Collecteurs et de leur outillage, et notamment de l'historique de collecte. Il s'agit d'une session de déverminage d'un problème difficile, rencontré par des développeurs lors du $refactoring$ d'un outil nommé *Pillar* [Arloing et al., 2016]. Les développeurs à l'origine de ce problème ne sont pas les développeurs ayant développé *Pillar*, et nous n'avons nous-mêmes aucune connaissance préalable de l'outil. Le simple ajout d'une variable d'instance et de deux méthodes d'accès en lecture et écriture dans une des classes de *Pillar* provoque l'échec d'un ensemble de tests unitaires, sans liens apparents avec la modification. Ce problème est considéré comme difficile à résoudre [Dupriez et al., 2017], car il n'y a pas de lien évident entre le contexte dans lequel il apparaît et ses symptômes.

Cette expérimentation consiste à utiliser les Collecteurs pour tenter de traquer l'erreur et de formuler des hypothèses sur les raisons du problème. L'utilisation des

Collecteurs a permis de comprendre le problème et d'en proposer une correction. La source de ce problème était toujours inconnue avant notre expérimentation. Nous décrivons et analysons la session de déverminage du problème *Pillar* avec les Collecteurs.

5.6.1 Description du problème

Le problème "Pillar Bug" est une erreur rencontrée par l'outil *Pillar*, un générateur de document à partir d'une syntaxe de type *mark-up* mise en œuvre avec le langage Pharo. L'outillage de *Pillar* est sophistiqué, et nécessite une base de plus de 3000 tests unitaires pour être correctement testé et validé. L'introduction d'accesseurs en lecture et écriture d'une nouvelle variable d'instance dans une classe fait échouer un ensemble restreint de ces tests, sans rapport aucun avec la modification. La description précise ainsi que les étapes de reproduction du problème sont publiquement accessibles [5]. Une nouvelle variable d'instance nommée *disabledPhases* est ajoutée à une classe de configuration. Deux accesseurs sont introduits : *disabledPhases* pour la lecture de la variable d'instance, et *disabledPhases*: pour la modification de la variable. Par défaut, si la variable *disabledPhases* n'a pas été initialisée, son accesseur l'initialise à un tableau vide. Avant l'introduction de ces accesseurs, l'exécution de toute la suite unitaire de Pillar se termine sans aucune erreur. Après leur introduction, plusieurs tests unitaires échouent. Au premier abord, l'erreur semble complètement indépendante de la modification effectuée.

Le problème est également décrit par Dupriez et al. [Dupriez et al., 2017]. L'erreur apparaît dans une méthode *actionOn*: d'une classe *PREPubMenuJustHeaderTransformer*. Un accesseur *outputType* d'une configuration renvoie une valeur nulle, ce qui provoque l'échec du test. Seul un ensemble très réduit de tests sont en échec, et cette méthode, sa classe et cet accesseur provoquant le problème n'ont aucun lien évident avec la modification apportée – c'est-à-dire l'ajout des deux accesseurs à la nouvelle variable *disabledPhases* dans la classe de configuration de *Pillar*.

Difficulté d'utilisation des outils de débogage traditionnels. Cette description du problème est plutôt abstraite. Il s'agit cependant de la situation dans laquelle nous nous retrouvons après l'introduction des deux accesseurs et d'une variable d'instance : des tests en échec, une erreur sans aucun lien avec cette modification triviale, et trop peu de connaissance de *Pillar* pour savoir où commencer le déverminage. Lors d'une première investigation, les techniques de déverminage traditionnelles n'ont pas permis de trouver la source du problème. L'exploration de la pile d'exécution au moment du *bug* ne permet pas de trouver d'indice sur l'origine du problème, ni la relation entre ce dernier et la modification apportée

5. https ://github.com/guillep/pillar-bug

au programme. Les points d'arrêt dans l'accesseur en lecture ajouté au programme (*disabledPhases*) sont difficiles à utiliser. L'accesseur est appelé de nombreuses fois au cours de l'exécution du test, et retourne constamment la même valeur qui, autant que nous puissions en juger, semble correcte − ou tout du moins n'est pas suspicieuse. L'insertion de traces permet de compter le nombre de passages dans cet accesseur. Ce dernier est appelé 42 fois dans une boucle, et tous ces appels, sauf le dernier, retournent le même résultat (un tableau de deux éléments). Nous pouvons placer un point d'arrêt conditionnel dans cette méthode, pour stopper le programme lorsque la valeur retournée change, mais aucun indice évident ne permet d'expliquer pourquoi cette valeur est différente des 41 autres. L'exploration du contexte et de la pile d'exécution à ce moment n'est d'aucune aide. En particulier, plus la pile d'exécution est explorée loin dans le passé, plus il est difficile de visualiser et de comprendre l'impact du code et du contexte inspectés sur le résultat final. L'accesseur en écriture n'est lui appelé qu'une seule fois et initialise la variable d'instance avec un tableau à deux éléments (mentionné ci-dessus) et qui semble être la valeur attendue. Les difficultés d'utilisation de ces outils pour ce problème proviennent principalement du manque de connaissance de *Pillar*, et de la difficulté à les utiliser au bon moment pour observer un état intéressant du programme dans le cadre de ce *bug*.

5.6.2 Observation du programme avant la modification problématique

Nous inspectons rapidement le programme avant l'introduction des accesseurs problématiques, afin de se représenter une exécution correcte du test en erreur. Nous remarquons que les accesseurs, bien que n'existant pas, sont tout de même appelés dans le code source du projet. Ces appels sont annotés comme *non implémentés* par l'environnement de développement. Cet appel à l'accesseur *disabledPhases* est illustré à la Figure 5.20. Nous définissons un Collecteur pour historiser les valeurs issues de l'exécution de cet appel (surligné en orange). Lors d'une exécution *saine*, plusieurs collectes sont effectuées, c'est-à-dire que le programme exécute plusieurs fois le code instrumenté. À chaque passage dans ce flot de contrôle, le résultat est archivé. Cette expression renvoie systématiquement la même valeur, qui est un tableau contenant deux chaînes de caractères : $'sections'$ et $'justKeepHeaders'$. Lors d'une exécution *saine*, le résultat de l'évaluation de cette expression semble donc être un invariant.

Une observation rapide du code montre que l'accesseur *disabledPhases* n'étant pas implémenté, la réception de ce message entraîne une exception $doesNotUnderstand$. Dans la classe de configuration impactée, le comportement du $doesNotUnderstand$ est instrumenté pour capturer certains messages (par exemple

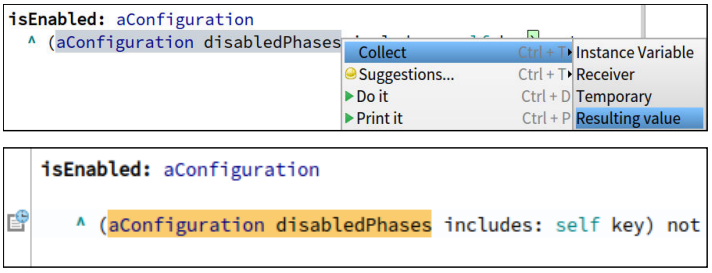


FIGURE 5.20 – Point de collecte sur l'appel à l'accesseur *disabledPhases* : chaque fois que le programme exécutera ce flot de contrôle précis, le résultat de l'évaluation de l'expression sélectionnée sera collecté.

disabledPhases) et les rediriger vers un dictionnaire contenant les propriétés de la configuration. Une configuration peut également avoir une configuration parente. Si une propriété recherchée n'est pas trouvée dans les propriétés de la configuration courante, par exemple *disabledPhases*, la propriété est recherchée dans sa configuration parente jusqu'à ce que la propriété soit trouvée ou qu'il n'y ait plus de configuration parente auquel cas *nil* est retourné par le *doesNotUnderstand*.

5.6.3 Observation du programme après introduction de l'erreur

Après introduction des accesseurs dans le code source, le test sort en erreur lors de son exécution. La consultation de l'historique (Figure 5.21) montre que lors de la dernière exécution il y a eu variation de la valeur collectée. L'invariant a été *cassé*, et remplacé par un tableau vide. Il s'agit d'une différence évidente entre l'exécution dite *saine* du programme et l'exécution problématique, et constitue une première piste d'investigation.

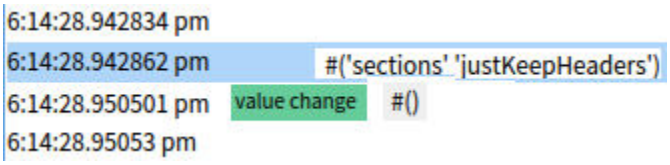


FIGURE 5.21 – Variation des objets collectés (montage) : l'historique du Collecteur indique par un libellé que la valeur a changé. L'objet collecté par l'expression instrumentée ne doit pas varier lors d'une exécution *saine*.

L'historique présenté par la Figure 5.21 contient les résultats de l'évaluation de l'expression instrumentée, mais aussi de ses sous-expressions. Ainsi lorsque le Collecteur collecte le résultat de l'envoi du message *disabledPhases* à l'objet *aConfiguration* (Figure 5.20), il collecte également le résultat de l'évaluation de la sous-expression *aConfiguration*. Nous pouvons donc comparer non seulement le résultat problématique de l'expression globale, mais également les valeurs et les états des objets et des évaluations utilisés dans cette expression. La comparaison des configurations avant et après le changement de l'invariant montre que ça n'est pas l'état de la configuration qui a été modifié, mais que les deux objets sont différents. La configuration d'origine est donc remplacée au cours de l'exécution du test, par une autre qui possède un état inattendu.

5.6.4 Traque du problème et correction

Cette nouvelle piste consiste à chercher à quel moment la configuration avec un état correct est remplacée par une nouvelle configuration avec un état incorrect. Nous décidons de collecter tous les objets de cette classe de configuration particulière lors de leur création. Nous introduisons à cet effet une méthode *initialize* dans notre classe de configuration pour pouvoir facilement collecter son résultat, c'est-à-dire chaque nouvelle instance de notre configuration. Nous nous assurons que l'ajout de cette méthode n'a pas d'impact sur le résultat du test, en exécutant toute la suite de tests dans une version de l'application *saine*, sans introduction des accesseurs problématiques.

Après une nouvelle exécution du test, l'historique de création de configurations montre plusieurs instanciations de la classe de configuration. L'inspection de la première configuration collectée permet d'observer un état correct des *disabledPhases* et qui correspond à l'invariant. Les autres collectes ont récupéré des configurations avec un état incorrect, c'est-à-dire un tableau vide ne correspondant pas à l'invariant recherché. Pour chaque collecte, il est possible de consulter la pile d'exécution ayant mené à l'exécution de l'expression instrumentée.

```
1 configuration
2     ^ super configuration
3     "we disable these phases as they pollute the tests"
4         disabledPhases: #('sections' 'justKeepHeaders');
5         yourself
```

FIGURE 5.22 – Méthode appelée dans la pile d'exécution de la première configuration collectée : l'invariant *disabledPhases* y est correctement initialisé.

La pile d'exécution de la première configuration collectée montre qu'il s'agit de celle créée lors de l'initialisation du test. Elle est notamment paramétrée via la méthode *configuration* qu'on retrouve dans la pile et qui initialise l'invariant des *disabledPhases* (Figure 5.22). La pile d'exécution de la seconde configuration collectée montre que cette dernière est créée par un tout autre *chemin* (Figure 5.23). En particulier, l'instanciation de la configuration est effectuée à partir d'une méthode *parseFile:*. Cette dernière instancie une nouvelle configuration, et lui donne comme parent la configuration précédente – dans le cas présent la configuration d'origine. Cette nouvelle configuration n'est pas initialisée et notamment aucune propriété ni aucune variable d'instance de cette configuration n'est modifiée.

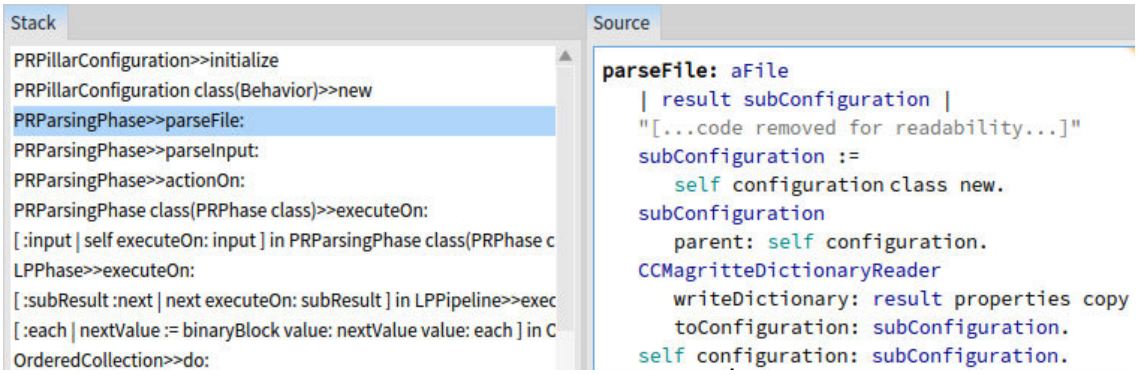


FIGURE 5.23 – Pile d'exécution amenant à la collecte de la seconde configuration instanciée. La nouvelle configuration instanciée devient fille de la configuration d'origine, sans copier son état.

Nous pouvons alors formuler une hypothèse. Dans le cas non problématique avant introduction des accesseurs dans la classe de la configuration, chaque envoi des messages *disabledPhases* et *disabledPhases*: provoque une exception de type *doesNotUnderstand*. Le comportement de cette dernière est instrumenté pour rechercher dans les propriétés de la configuration la propriété *disabledPhases* (Figure 5.24). Dans le cas de figure que nous observons dans la pile de la Figure 5.23, la nouvelle configuration créée n'est pas initialisée. Chaque tentative de recherche de la propriété *disabledPhases* aboutira dans le *doesNotUnderstand*, qui recherchera la propriété dans la configuration parente. Ce comportement, déjà décrit à la section 5.6.2, explique que la propriété soit toujours correctement initialisée et sa valeur invariante.

Dans le cas problématique dû à l'ajout des accesseurs, les envois de messages *disabledPhases* et *disabledPhases*: ne provoquent plus de *doesNotUnderstand*, car le message est désormais compris par l'objet de configuration. Lorsque la nouvelle

configuration reçoit ce message, elle y répond immédiatement par l'exécution de son accesseur qui renvoit sa variable d'instance initialisée à un tableau vide par défaut. Il n'y a plus de recherche de la propriété dans les configurations parentes, puisque le comportement instrumenté du *doesNotUnderstand* n'est plus exécuté. La configuration n'est plus capable de répondre à ce message de manière cohérente, et brise l'invariant paramétré (Figure 5.24).

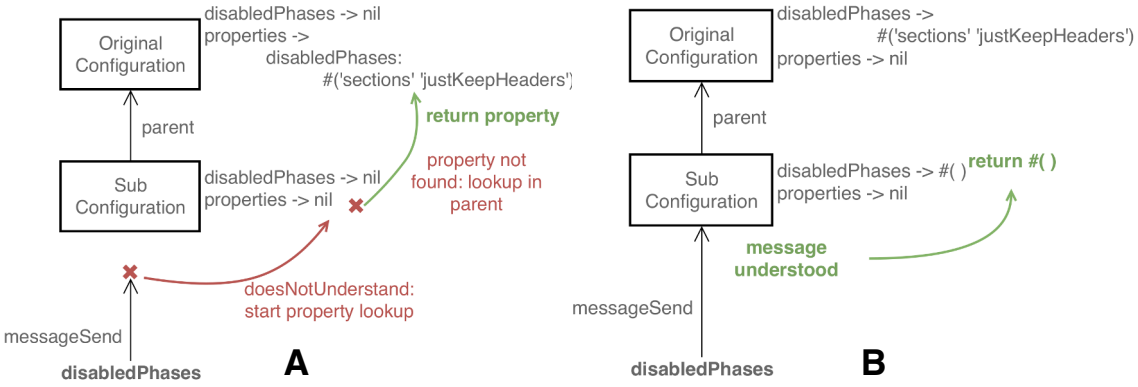


FIGURE 5.24 – *Lookup* des propriétés des configurations Pillar. A) La propriété *disabledPhases* est retrouvée au travers d'un lookup parallèle après un échec du lookup standard. B) Après introduction des accesseurs, le lookup standard n'échoue plus et le lookup de propriété n'est plus effectué.

En outre, le commentaire dans le code de la Figure 5.22 indique que la conception du test exige ce paramétrage spécifique de la propriété *disabledPhases*. L'erreur provient de la sémantique particulière placée sur le *doesNotUnderstand* des configurations, qui est cassée par l'introduction des accesseurs dans la classe de la configuration. Nous expérimentons une simple correction (Figure 5.25), qui consiste à mettre à jour l'état de la variable *disabledPhases* de chaque nouvelle configuration à partir de sa configuration parente. Le test unitaire ne provoque plus d'erreurs et la suite complète (plus de 3000 tests) se termine également avec succès.

```
1 subConfiguration disabledPhases:
2     self configuration disabledPhases
```

FIGURE 5.25 – Correction expérimentée à la suite de la création d'une nouvelle sous-configuration : copie des *disabledPhases* à partir de la configuration parente.

5.6.5 Discussion

Il s'agit d'un problème difficile, car la sémantique mise en place par Pillar est non triviale pour une personne externe au projet. Une modification en apparence anodine provoque ici une erreur, dont la difficulté réside dans le fait que la relation entre la modification apportée au système et le symptôme apparent de l'erreur sont très éloignés l'un de l'autre [Eisenstadt, 1997].

L'utilisation des Collecteurs et de leur outillage a permis de trouver l'origine de l'erreur et de la comprendre en quelques minutes. L'outillage ne fait qu'utiliser le modèle du langage pour mettre en œuvre les avantages des dévermineurs dits *back-in-time* ou *omniscients* (chapitre 1, section 1.2.2). La visualisation des enregistrements de l'exécution du programme permet de suivre l'évolution des objets et leurs interactions. Mais l'apport des Collecteurs réside ici dans l'approche utilisée, dans laquelle le développeur ou la développeuse utilise sa connaissance du programme et l'intuition qui en résulte pour sélectionner manuellement les expressions à instrumenter. Le résultat de l'expression sera collecté chaque fois qu'elle sera exécutée, ainsi que tous les objets résultant de l'évaluation des sous-expressions de cette dernière. Pour chaque collecte, un certain nombre d'éléments contextuels à l'exécution sont enregistrés pour aider le développeur à analyser le résultat de l'opération. Par exemple, les éléments suivants ont été cruciaux lors de cette investigation :

— L'observation d'un état invariant cassé entre une exécution *saine* et une exécution problématique nous a aidés à comprendre que l'objet configuration variait au cours du temps, et qu'il fallait traquer les configurations instanciées après l'originale et dont l'état n'était pas cohérent.

— L'accès à la pile d'exécution avec ses contextes amenant à la collecte d'une configuration, résultat de l'exécution de l'expression d'instanciation de l'objet, nous a permis de trouver presque immédiatement la cause de l'état incohérent.

Ces étapes reposent cependant sur une analyse préalable du code du programme étudié, afin d'obtenir une connaissance et une compréhension suffisante pour formuler des hypothèses. L'outil ne donne pas de solution au problème, mais complète cette connaissance du programme en donnant la possibilité d'instrumenter rapidement et intuitivement les expressions suspectes. La facilité de mise en place de Collecteurs et leur intégration à l'environnement de développement sont à cet effet très importantes. Le filtrage par le développeur-euse des expressions collectées, suivant son intuition et sa compréhension du programme, permet de cibler précisément le périmètre du code instrumenté et dont le résultat – des objets – sera à analyser.

5.7 Discussion générale et évaluation

Nous avons présenté la mise en œuvre des Collecteurs, et différents cas d'utilisation illustrant leur capacité à regrouper de manière non-anticipée des objets dans un programme en cours d'exécution. Dans cette section, nous évaluons les Collecteurs au regard des propriétés que nous recherchons pour l'adaptation non-anticipée de comportement, et que nous avons définies dans le chapitre 1. Nous discutons ensuite la mise en œuvre des collecteurs et leur outillage d'un point de vue général.

5.7.1 Évaluation

● Totalement supporté ◑ Partiellement supporté ○ Non supporté

	Granularité objet	Minage d'objets	Réversibilité	Identité préservée	Flexibilité	Non-intrusivité	Paradigme objet
Collecteurs	○	●	●	●	●	●	●

TABLE 5.1 – Positionnement des Collecteurs et de leur mise en œuvre au regard de nos contraintes pour l'adaptation non-anticipée.

11

La table 5.1 présente l'évaluation des collecteurs au regard des contraintes pour l'adaptation non-anticipée. La propriété principale atteinte par les Collecteurs est celle du minage d'objets complètement non-anticipée. Un collecteur peut être installé et désinstallé à la demande au travers de l'API et de l'outillage (réversibilité). Enfin, l'utilisation des Collecteurs pour déboguer un programme n'impose pas de contraintes particulières sur le programme ou sur la manière de les mettre en œuvre.

Les propriétés de préservation d'identité, de non-intrusivité et du maintient du paradigme objet sont hérités de Reflectivity, composante essentielle sur laquelle repose la mise en œuvre des Collecteurs.

Bien qu'il soit possible de restreindre un collecteur à un objet précis, nous ne considérons pas que cela remplisse la propriété d'adaptation à granularité objet. Le comportement de l'objet n'est, en effet, pas affecté par le collecteur.

5.7.2 Discussion

L'utilisation des Collecteurs et de leur outillage n'invalide pas l'utilisation d'un déverminer classique, qui peut être complémentaire. Par exemple, l'utilisation de points d'arrêts à des endroits clés dans le code permet de s'arrêter et d'inspecter le programme. Il n'y a cependant pas de garanties que l'arrêt se produise avant le changement d'état problématique, ni qu'il se produise dans des temps raisonnables lors d'une exécution pas à pas manuelle dans le déverminer. Si le point d'arrêt – ou l'exécution pas à pas – manque le point critique dans l'exécution du programme où un état est modifié, par exemple ici les *disabledPhases*, le développeur ou la développeuse doit recommencer sa session de déverminage. C'est un problème classique particulier à cette technique [Beller et al., 2018]. Le point d'arrêt ne permet pas non plus de visualiser simplement les changements d'états d'un objet entre différentes exécutions et d'en comparer certains aspects spécifiques à leurs contextes – par exemple les piles d'exécutions. En outre, les instrumentations pour observer un objet particulier peuvent être complexes et nécessiter l'insertion de code à différents endroits du programme. L'aspect non-intrusif des Collecteurs, hérité de sa couche de mise en œuvre Reflectivity, permet d'éviter de telles insertions et d'éviter la pollution du code par des instructions de déverminage.

Au regard de la littérature, les Collecteurs sont très proches du *Dynamic Query-based Debugging* [Lencevicius et al., 1999, Lencevicius, 2000, Lencevicius et al., 2003] (chapitre 1, section 1.2.2). Le *Query-based-Debugging* permet d'obtenir des groupes d'objets à partir de requêtes spécifiées par l'utilisateur-rice. La finalité est similaire mais les perspectives sont différentes. Tandis que le *Query-based Debugging* explicite des groupes d'objets à partir de contraintes sur leur domaine − typiquement des conditions sur leur état − les Collecteurs utilisent la structure du programme pour déterminer quels objets récupérer (par exemple une variable ou le receveur d'un message). Une fois qu'un objet est pré-sélectionné par un collecteur, c'est à dire que les contraintes sur la structure du programme sont satisfaites, des contraintes sur le domaine de l'objet peuvent être exprimées pour confirmer la collecte. Les développeur-euse-s raisonnent alors sur le code source, qui est l'abstraction principale du programme en cours d'exécution. Les Collecteurs permettent de connecter cette abstraction statique à des objets du programme en cours d'exécution, facilitant ainsi le déverminage.

Comme précisé à la section 5.2.2, la consommation mémoire supplémentaire et les ralentissements potentiels dus à l'utilisation des Collecteurs n'ont pas été étudiés. Dans le contexte de l'exécution de tests unitaires, dont seulement un ensemble restreint fait appel à une méthode instrumentée par un collecteur, aucun ralentissement n'est visible du point de vue d'un-e développeur-euse. Aucune surconsommation mémoire apparente n'a affecté le système. Mettre ces problématiques de coté était important pour expérimenter librement les fonctionnalités des Collecteurs. Il

s'agit cependant d'un aspect qui doit être étudié, et qui constitue probablement une faiblesse de la mise en œuvre présentée. La question de l'impact de la collecte sur l'exécution d'une suite de tests comme celle de *Pillar* reste non évaluée. *Pillar* contenant plus de 3000 tests unitaires, l'impact sur le fonctionnement du système pourrait être non négligeable en termes de performances et de consommation mémoire si tous les tests faisaient appel à une méthode avec un collecteur. Que les Collecteurs et leurs outils puissent être utilisés en pratique reste donc à démontrer.

De la même manière, dans le cas de l'illustration du *bug Pillar*, il est tout à fait possible qu'un-e développeur-euse avec une connaissance plus fine de *Pillar* puisse trouver rapidement le problème avec des outils de déverminage classiques, tels que les points d'arrêt. Pour positionner précisément l'utilité des Collecteurs en pratique, il manque à cette évaluation une étude contrôlée avec différents types de développeurs. L'objectif serait de déterminer à quels développeurs l'outil profite le plus, pour quels types de problèmes, et de mettre les résultats en relation avec les études déjà menées dans la littérature concernant les outils achevant des objectifs similaires (déverminneurs *omniscients*, etc.).

En terme d'outillage, une intégration avec le déverminneur de l'environnement de développement (de Pharo, Python...) permettrait d'utiliser les Collecteurs suivant un autre point de vue. Un Collecteur pourrait être directement lié à un point d'arrêt, et provoquer l'arrêt du programme et l'ouverture d'un débogueur juste après une collecte − soumise à condition ou non. L'utilisateur-rice pourrait décider d'utiliser tout ou partie des objets collectés pour rejouer une partie de l'exécution, et effectuer du pas à pas dans le débogueur. Ce-tte dernier-e pourrait alors redémarrer l'exécution de sa méthode autant de fois que voulu, avec la garantie de reproduire les résultats des expressions qui sont rejouées. D'un point de vue mémoire et performances, la perspective est différente dans la mesure où le programme serait interrompu par le débogueur lors de la première collecte − ce qui limite très fortement l'accumulation en mémoire de copies d'objets collectés − et l'aspect performance est significativement moins critique lors d'un pas-à-pas dans le déverminneur que lors d'une exécution en environnement de production.

5.8 Conclusion

Dans ce chapitre, nous avons décrit les Collecteurs et leur outillage pour le minage d'objets non-anticipé dans un programme en cours d'exécution. Les Collecteurs permettent de spécifier dynamiquement des points de collecte dans le flot de contrôle d'un programme, à partir desquels les objets issus de l'évaluation des expressions ciblées sont regroupés et présentés à l'utilisateur-rice. Nous avons décrit des contraintes de mise en œuvre, et comment nous les avons abordées avec Reflectivity et son extension. Nous avons alors présenté un modèle de mise en œuvre

pour le langage Pharo. Cette mise en œuvre comprend une interface utilisateur et un outillage intégré à l'environnement de développement de Pharo. Pour valider l'aspect patron de langage, nous avons également décrit une mise en œuvre − moins complète − dans le langage Python et basée sur un portage de Reflectivity.

Trois cas d'utilisation ont été présentés. Premièrement, une illustration de l'utilisation des Collecteurs pour produire des traces d'objets conditionnelles dans un programme en cours d'exécution. Secondement, la fonctionnalité *replay* des Collecteurs est utilisée pour reproduire un problème dont l'occurrence est non-déterministe. Un objet provoquant un *bug* est collecté et utilisé pour rejouer l'exécution afin d'éliminer cet aspect non-déterministe. Enfin, nous présentons une session de débogage d'un problème dans l'outil *Pillar*, et dans laquelle les historiques de collecte permettent de trouver la source d'un *bug* dont les symptômes sont éloignés de cette source.

Les Collecteurs ont été évalués au regard des propriétés recherchées pour l'adaptation de comportement, et montrent un positionnement fort sur la propriété de minage non-anticipé d'objets dans les programmes en cours d'exécution.

Nous avons finalement discuté de l'utilisation des Collecteurs et de leur outillage, et soulevé des points d'amélioration possibles. Les pistes pour des travaux futurs sont notamment l'intégration des Collecteurs dans le débogueur Pharo, l'évaluation de leur outillage dans une étude de terrain auprès de développeurs professionnels et l'investigation d'optimisation concernant les performances et l'impact sur la mémoire des programmes instrumentés.

Chapitre 6

Une infrastructure pour l'adaptation non-anticipée de comportement appliquée au déverminage d'applications en cours d'exécution

---

6.1 Infrastructure et outillage pour l'adaptation non-anticipée dans les programmes en cours d'exécution . . . 207
6.1.1 *Debug-Scopes* : périmètres dynamiques de l'adaptation . . . 208
6.1.2 Définition d'un périmètre, de ses activateurs et de ses collecteurs 212
6.1.3 Définition d'adaptation de comportement . . . 213
6.1.4 Visualisation du comportement adapté . . . 217
6.1.5 Compatibilité avec les outils natifs de Pharo . . . 218
6.2 Cas d'utilisation sur un objet connecté : l'application *Sensor Monitoring App* et le *chat-bot IotZ* . . . 218
6.3 Scénario 1 : déverminage non-anticipé en cours d'exécution . . . 221
6.3.1 Détection du problème . . . 222
6.3.2 Correction du problème par une adaptation de comportement 223
6.3.3 Discussion . . . 226
6.4 Scénario 2 : utilisation non-anticipée de la caméra pour la prise de photographies de surveillance . . . 227
6.4.1 Adaptation du programme pour utiliser la caméra . . . 227
6.4.2 Discussion . . . 233
6.5 Scénario 3 : extension d'interface utilisateur pour du déverminage . . 234
6.5.1 Adaptation de l'interface du *chat-bot* . . . 235
6.5.2 Discussion . . . 238
6.6 Discussion générale et évaluation . . . 239

6.6.1 Évaluation 239
6.6.2 Bénéfices des périmètres d'adaptation 240
6.6.3 Problèmes et limitations 240
6.7 Conclusion 243

Dans le chapitre 4, nous avons décrit la mise en œuvre du patron de langage Lub, sous la forme d'une extension du langage Pharo. Lub permet d'adapter des objets uniques de manière non-anticipée, en leur permettant de se comporter partiellement comme des instances d'autres classes. Dans le chapitre 5, nous avons décrit la mise en œuvre du patron de langage des Collecteurs, sous la forme d'une extension de Pharo. Les Collecteurs permettent de regrouper dynamiquement des objets dans un programme en cours d'exécution.

Ce chapitre présente les *Debug-Scopes*, une infrastructure qui combine Lub et les Collecteurs au travers d'un outillage pour l'adaptation non-anticipée dans les programmes en cours d'exécution. Durant l'exécution du programme, les objets regroupés par les Collecteurs sont présentés à Lub pour être soumis à l'adaptation de comportement.

Nous décrivons l'infrastructure des *Debug-Scopes*, et présentons une application sur un cas de déverminage à distance d'un objet connecté. Ce cas d'utilisation concerne un *chat-bot* qui surveille des données environnementales, et fournit une interface de communication avec des utilisateur-rice-s expert-e-s. Nous divisons cette évaluation en trois scénarios, qui illustrent chacun un aspect de l'adaptation non-anticipée appliquée au déverminage. Le premier montre la mise en place d'un correctif suite à une erreur pendant l'exécution du programme. Le second montre l'ajout d'une fonctionnalité d'observation spécifique en tirant partie de la caméra présente sur l'objet connecté. Le dernier scénario montre comment l'adaptation permet d'étendre l'interface du *chat-bot* sur une demande non-anticipée des utilisateur-rice-s du programme. Chaque scénario est suivi d'une courte discussion spécifique à l'aspect étudié, puis l'infrastructure et son outillage sont discutés plus en détail dans l'avant-dernière section, où nous faisons le bilan des avantages et des limitations de la solution.

6.1 Infrastructure et outillage pour l'adaptation non-anticipée dans les programmes en cours d'exécution

Un *Debug-Scope*, ou *périmètre d'adaptation*, définit le périmètre d'une adaptation d'un programme en cours d'exécution. L'adaptation est considérée ici au sens large, et peut concerner plusieurs adaptations de comportement différentes appliquées au même programme. Un périmètre définit, pour un programme en cours d'exécution, quels sont les objets qui seront adaptés (collecteurs), comment ils seront adaptés (adaptations) et à quel moment (gardes des adaptations). Les périmètres d'adaptation — que nous nommerons plus simplement *périmètres* dans le reste de la thèse — sont directement mis en œuvre au travers d'un outillage intégré dans Pharo et reposant sur les implémentations de Lub et des Collecteurs. Cette section décrit l'infrastructure des *Debug-Scopes* et son outillage, nommément le navigateur de périmètres et son intégration dans l'environnement Pharo.

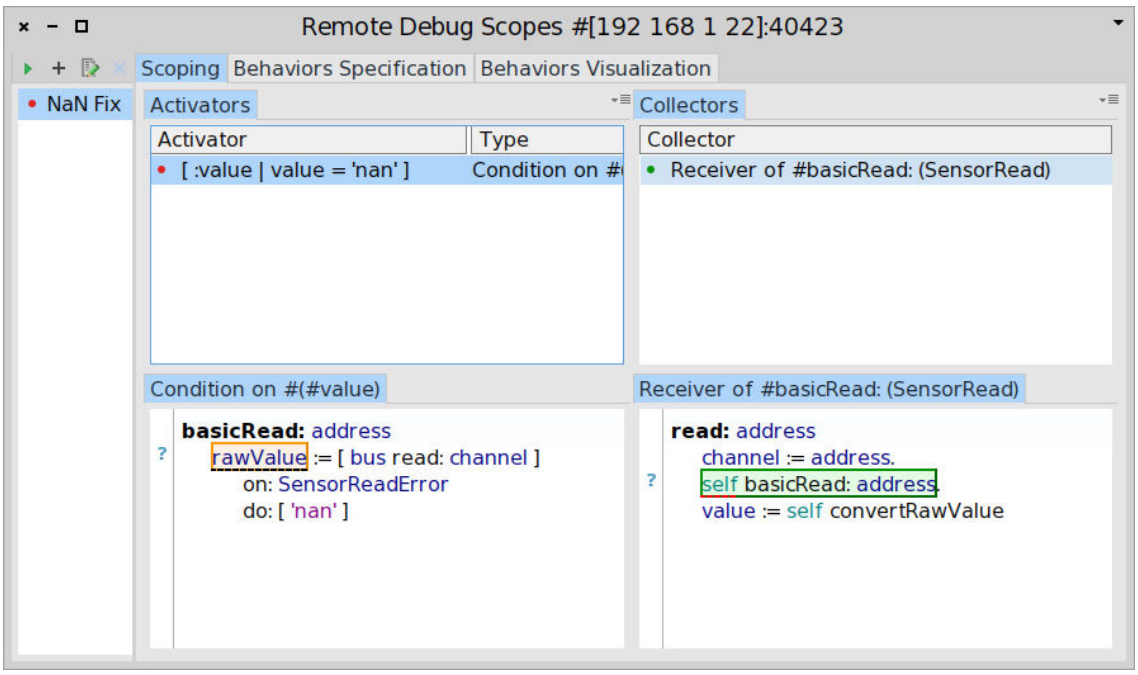


FIGURE 6.1 – Navigateur de périmètres (premier onglet) : le périmètre *NaN Fix* définit un activateur (garde d'adaptation) et un collecteur. Colonne de gauche : le périmètre. Colonnes du milieu, pour le périmètre sélectionné : les activateurs, gardes d'adaptations (première colonne) et les collecteurs (dernière colonne).

6.1.1 *Debug-Scopes* : périmètres dynamiques de l'adaptation

Un périmètre peut être créé et modifié dynamiquement pendant l'exécution d'un programme. Le navigateur de périmètres (Figure 6.1) est directement intégré à Pharo, et peut interagir avec un programme Pharo s'exécutant dans le même environnement (cas du *Live Programming*) ou dans un environnement distant. Le navigateur utilise à cet effet l'outillage de *remote debugging* présent dans Pharo [Papoulias et al., 2015, TelePharo, 2018]. Un périmètre est défini par un nom, un état $actif$ ou $inactif$, un ensemble de conditions d'activation nommées $activateurs$, un ensemble de collecteurs qui regroupent dynamiquement les objets à adapter et une adaptation dédiée pour chaque collecteur. Les paragraphes suivants décrivent le modèle de l'infrastructure des périmètres, illustré par la Figure 6.2.

Scope. Une instance de *Scope* est un périmètre d'adaptation. Ce dernier possède un nom et un ensemble dédié d'activateurs, de collecteurs et d'adaptations. Plusieurs périmètres peuvent être définis et représentent des variations comportementales indépendantes les unes des autres. Ils sont visibles dans la première colonne à gauche dans la Figure 6.1. L'activation d'un périmètre déclenche l'application des adaptations sur les objets agrégés par ses collecteurs. Dans cette mise en œuvre, un seul périmètre peut être actif à la fois − c'est-à-dire qu'une seule variation comportementale peut être appliquée au programme à un instant t.

RBProgramNode, MetaLink, NodeReification. Le modèle des périmètres utilise un certain nombre d'entités du langage, et notamment de la couche réflexive de Pharo, nommément Reflectivity [Denker, 2008]. La classe *RBProgramNode* représente un noeud de l'arbre de syntaxe abstraite (AST). L'AST représente le programme sous forme de syntaxe abstraite, dans notre cas un tel nœud représente une expression du programme. L'AST peut être annoté par des instances de *MetaLink*, qui déclenchent un comportement spécifique lorsqu'un nœud annoté est atteint par l'exécution du programme. Le fonctionnement des metalinks a déjà été décrit dans le chapitre 4 à la section 4.2.1. Enfin, une instance de *NodeReification* représente une réification nommée fournie par un metalink, lorsque ce dernier déclenche une instrumentation sur un nœud annoté. Par exemple, il peut s'agir du nom d'une variable d'instance, ou de l'objet receveur d'un message. La liste des réifications doit être spécifiée dans les instances de *MetaLink*.

Activator. Les activateurs sont des réifications des gardes d'adaptation (seconde colonne de la Figure 6.1). Ils constituent un ensemble de conditions évaluées séparément les unes des autres. Le périmètre est activé dès que toutes les conditions sont évaluées au littéral $true$, et désactivé dès qu'une des conditions est évaluée à $false$. Dans la Figure 6.1, un activateur est placé sur une affectation particulière

de variable d'instance $rawValue$, dans une méthode *basicRead*:. Chaque fois que le programme exécutera le code de cette méthode et affectera une nouvelle valeur dans la variable d'instance $rawValue$, la condition de l'activateur sera évaluée. La condition consiste à évaluer si la valeur affectée à la variable $rawValue$ est la chaîne de caractères $'nan'$. Si c'est le cas, le périmètre est activé. Deux types d'activateurs sont mis en œuvre :

Les activateurs conditionnels : ce sont les instances de la classe *ConditionalActivator*. Ils sont activés lorsqu'une condition spécifique est remplie. La condition de l'activateur est une instance de *Condition*. La condition peut prendre en argument des réifications fournies par les instances de *MetaLink*, et qui sont des instances de *NodeReification*. La condition est évaluée lorsque le programme en cours d'exécution atteint un certain point du flot de contrôle. Ce point du flot de contrôle est modélisé par un nœud de l'AST, instance de *RBProgramNode*. Si le programme ne passe plus par ce flot de contrôle, la condition n'est plus évaluée. Enfin, l'activateur conditionnel utilise une instance de *MetaLink*, qui déclenche l'évaluation de sa condition lorsque l'AST ciblé est atteint durant l'exécution du programme. L'activateur configure le metalink en fonction de sa condition pour obtenir des réifications comme arguments, et installe ce metalink sur le nœud de l'AST ciblé.

Les activateurs basés sur le flot de contrôle : ce sont les instances de la classe *ControlFlowActivator*. Ils sont activés lorsque le programme entre dans un flot de contrôle particulier et désactivés lorsque le programme sort de ce flot de contrôle. Typiquement, ils visent à restreindre l'activation d'un périmètre à l'exécution d'un morceau de code spécifique. En dehors de ce flot de contrôle, les adaptations sont inactives. Ce flot de contrôle est modélisé par deux nœuds de l'AST, qui représentent respectivement le début du flot de contrôle (*startNode*) et la fin du flot de contrôle ciblé (*stopNode*). L'entrée et la sortie de ce flot de contrôle modélisent l'activation et la désactivation de l'activateur. Deux instances de *MetaLink* sont utilisées et configurées par l'activateur, respectivement pour déclencher son activation lorsque *startNode* est atteint (*activationLink*) et pour déclencher sa désactivation lorsque *stopNode* est atteint (*deactivationLink).*

Adaptation. Les instances d'*Adaptation* sont des adaptations de comportement au sens Lub telles que décrites dans le chapitre 4. Une adaptation est liée à une classe et spécifie le comportement adapté tiré de cette classe, ainsi qu'un contrôle d'exécution (*before, instead, after*). Le contrôle définit comment est inséré du comportement adapté, avant, en remplacement ou après le comportement d'origine. La spécification des adaptations en pratique est décrite dans la section 6.1.3 suivante. Un périmètre peut définir plusieurs adaptations. Bien que les activateurs soient des

réifications des gardes d'adaptation, l'adaptation ne peut être appliquée tant que le périmètre n'est pas activé par ses activateurs. Concrètement, c'est donc le périmètre qui joue le rôle effectif de garde pour chaque instance d'*Adaptation* dont il est composé.

Collector. Les collecteurs utilisés par les périmètres d'adaptation sont les collecteurs étendus à l'adaptation (chapitre 3, section 3.3). La mise en œuvre utilisée est celle décrite dans le chapitre 5. Chaque collecteur est configuré avec une stratégie de conservation des objets par référence faible. Ils ne sont utilisés que pour obtenir des références d'objets à adapter pendant l'exécution du programme. Il n'y a donc pas d'interférence avec la gestion mémoire du programme en cours d'exécution, ni de surconsommation mémoire due à la sauvegarde d'objets dans le programme en cours d'exécution. Un collecteur est lié à une unique adaptation (qui peut composer du comportement), et lorsque le périmètre est activé cette adaptation est appliquée à tous les objets collectés par ce collecteur.

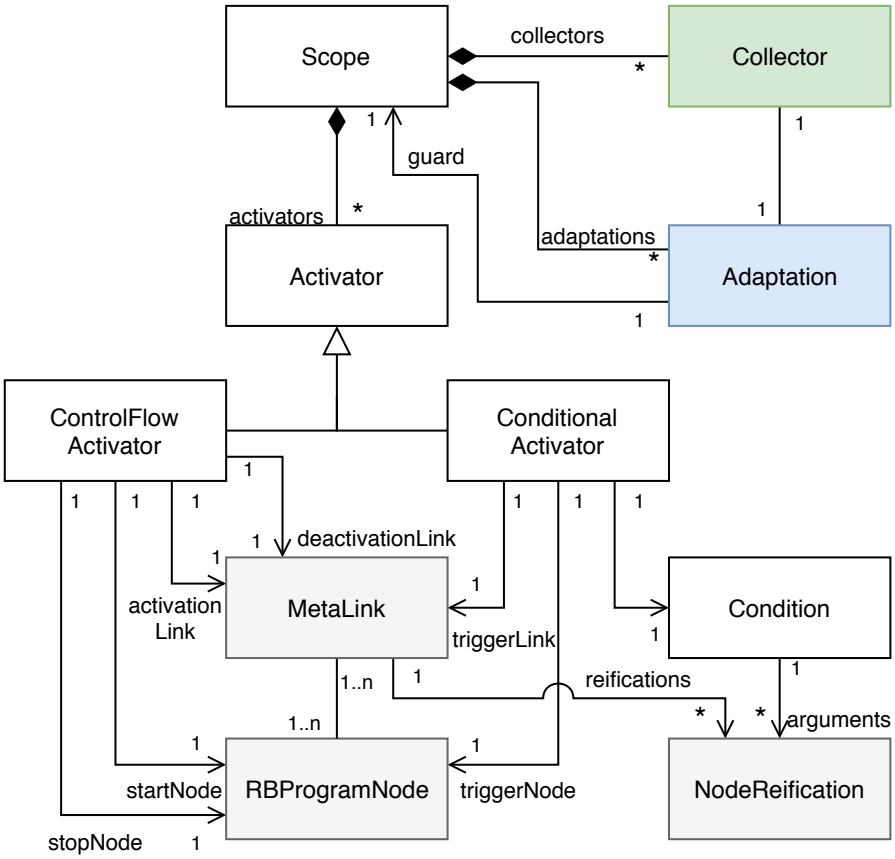


FIGURE 6.2 – Modèle de mise en œuvre des *Debug-Scopes*.

Adaptation non-anticipée à distance. Les périmètres d'adaptation peuvent être utilisés pour adapter un programme distant (Figure 6.3). Dans ce cas, l'utilisateur-rice interagit avec les périmètres distants de manière transparente : de son point de vue, il manipule directement les entités du programme en cours d'exécution (code source, adaptations, collecteurs, activateurs, objets...). La mise en œuvre des périmètres distants repose sur *TelePharo* [TelePharo, 2018], la suite de déverminage à distance fournie par Pharo. L'outillage des périmètres masque cette mécanique et donne l'impression à l'utilisateur-rice de manipuler directement les entités du programme distant. *TelePharo* fournit également un déverminer à distance qui permet d'intercepter les erreurs, et un navigateur de classe qui permet de naviguer dans le code source du programme. Par exemple lors de la spécification d'un collecteur (Figure 6.5), l'interaction s'effectue sur le navigateur de classes à distance de Pharo [TelePharo, 2018]. Le code affiché est alors une représentation textuelle du code distant en cours d'exécution.

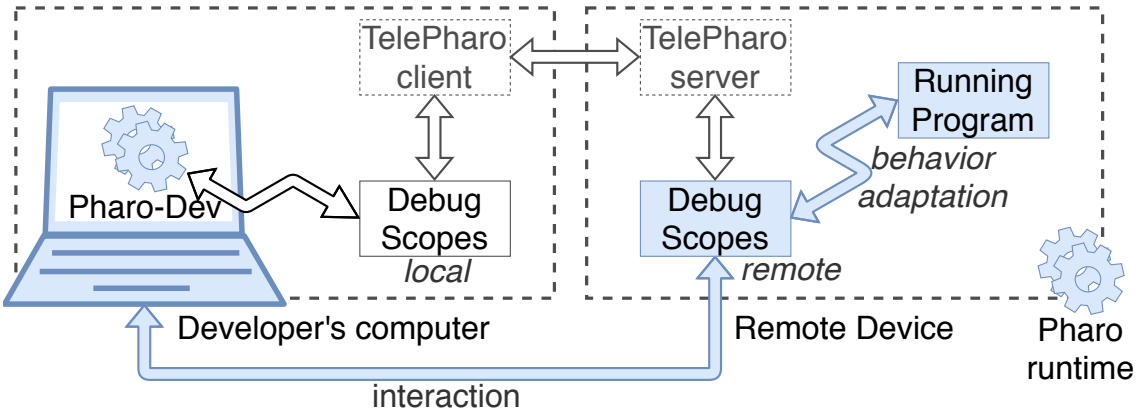


FIGURE 6.3 – Adaptation d'un programme distant : le développeur ou la développeuse visualise et interagit avec des périmètres d'adaptation comme s'il s'agissait directement de ceux du programme en cours d'exécution. Cette interaction à distance repose sur l'outillage *TelePharo*, la suite de déverminage à distance fournie par Pharo.

6.1.2 Définition d'un périmètre, de ses activateurs et de ses collecteurs

Les paragraphes suivants décrivent la définition d'un périmètre, telle qu'illustrée par la Figure 6.1.

Définition d'un périmètre. La définition d'un périmètre ne nécessite qu'une simple action de l'utilisateur-rice par un bouton de l'interface (bouton + en haut à gauche de la Figure 6.1). L'utilisateur-rice doit alors fournir un nom au nouveau périmètre. Lorsque la session d'adaptation s'effectue à distance, les périmètres et leurs composantes visibles dans le navigateur sont ceux présents dans le programme en cours d'exécution dans son environnement distant. C'est le cas de la Figure 6.1, où l'adresse IP de l'environnement distant est inscrite dans la barre de titre du navigateur.

Définition d'un activateur. La Figure 6.4 illustre la spécification d'un activateur. Cette action est disponible par le menu contextuel de Pharo, sur une sélection de code. Le menu permet de définir, pour un périmètre donné, des activateurs et des collecteurs. La sélection de code à partir de laquelle est spécifié l'activateur est donc définie sur le flot de contrôle du programme distant en cours d'exécution.

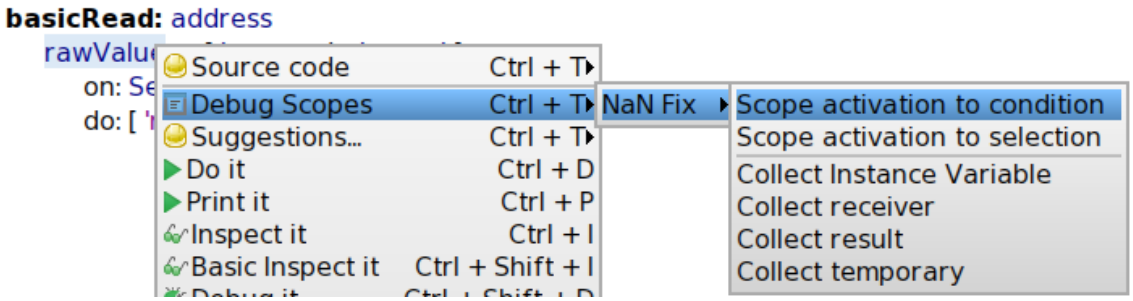


FIGURE 6.4 – Définition d'un activateur par interaction directe avec le code source.

La Figure 6.5 illustre la définition d'un activateur conditionnel au travers de l'outil ouvert après la sélection du code dans le flot de contrôle qui sera sujet à l'évaluation d'une condition (Figure 6.4). La colonne de gauche montre l'entité sélectionnée dans le flot de contrôle. La colonne de droite permet à l'utilisateur-rice de définir une condition d'activation dans un bloc de code, à partir des méta-données disponibles pour cette entité. Ces méta-données sont affichées à titre indicatif entre commentaires (*Available reifications*). Dans cet exemple, la méta-donnée *value* est utilisée pour représenter la valeur de l'entité sélectionnée dans le code source. Cette

méta-donnée représente donc la valeur stockée dans la variable $rawValue$, lors du passage du programme dans le flot de contrôle de la méthode *basicRead*: pendant son exécution. Plusieurs activateurs peuvent être définis sur le même flot de contrôle, sans restriction. En revanche, l'activation du périmètre dépend de l'évaluation indépendante de la condition de chaque activateur. Si une seule des conditions est évaluée au littéral $false$, le périmètre est désactivé.

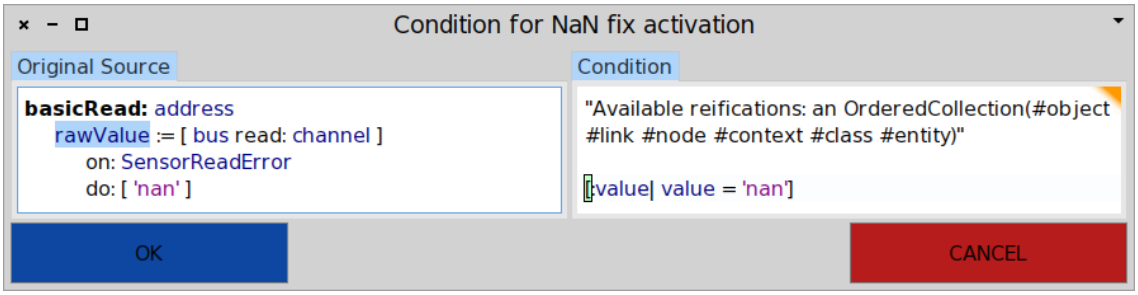


FIGURE 6.5 – Spécification d'un activateur conditionnel. À gauche, le point du flot de contrôle qui sera soumis à la condition, automatiquement sélectionné par l'outil en fonction de la sélection de l'utilisateur-rice. À droite, le bloc de code utilisateur qui définit la condition.

Définition d'un collecteur. La définition des collecteurs se fait suivant les possibilités fournies par leur mise en œuvre. Notamment, l'outillage de spécification des points de collecte est identique à celui déjà décrit dans le chapitre 5, et est également visible dans la Figure 6.4. L'outil de configuration des collecteurs n'est pas intégré au navigateur de périmètre, il n'est donc pas possible de spécifier des gardes de collecte. Dans le cas d'une session d'adaptation à distance, les points de collecte sont directement injectés dans le programme distant.

6.1.3 Définition d'adaptation de comportement

Le deuxième onglet du navigateur de périmètre (Figure 6.6) permet de spécifier une adaptation de comportement pour chaque collecteur. Lorsque le périmètre est activé lors de l'exécution du programme, il s'agit de l'adaptation qui sera appliquée à tous les objets du collecteur auquel cette dernière est associée.

La première case de la première colonne (*Object Collectors*) liste les collecteurs définis pour ce périmètre. La seconde case de la première colonne (*Debug Method*) permet de spécifier du comportement. La seconde colonne contient la liste des méthodes d'adaptation définies par l'utilisateur-rice (*Debug Behaviors*) ainsi que la manière dont ces méthodes modifient le comportement d'origine de l'objet adapté (*Controlled Behaviors*).

Debug Method. Les méthodes qui fournissent le comportement d'adaptation peuvent être spécifiées et modifiées dynamiquement. Si le périmètre est actif pendant un ajout ou une modification d'adaptation, le nouveau comportement n'est pris en compte qu'après la prochaine activation du périmètre.

Le nom des méthodes d'adaptation est libre, mais doit être unique. À cet effet, toutes ces méthodes sont automatiquement préfixées par la chaîne de caractères *debug*, indiquant leur caractère spécifique à l'adaptation.

L'édition du code de ces méthodes se fait hors du contexte d'exécution. Par exemple dans la Figure 6.6, la variable $rawValue$ est colorée en rouge, car elle n'est définie dans aucun contexte. Implicitement, lorsqu'un objet acquiert dynamiquement cette méthode, $rawValue$ devient une référence à la variable d'instance du même nom de cet objet. Si cette dernière n'existe pas, une erreur sera soulevée à l'exécution de la méthode adaptée.

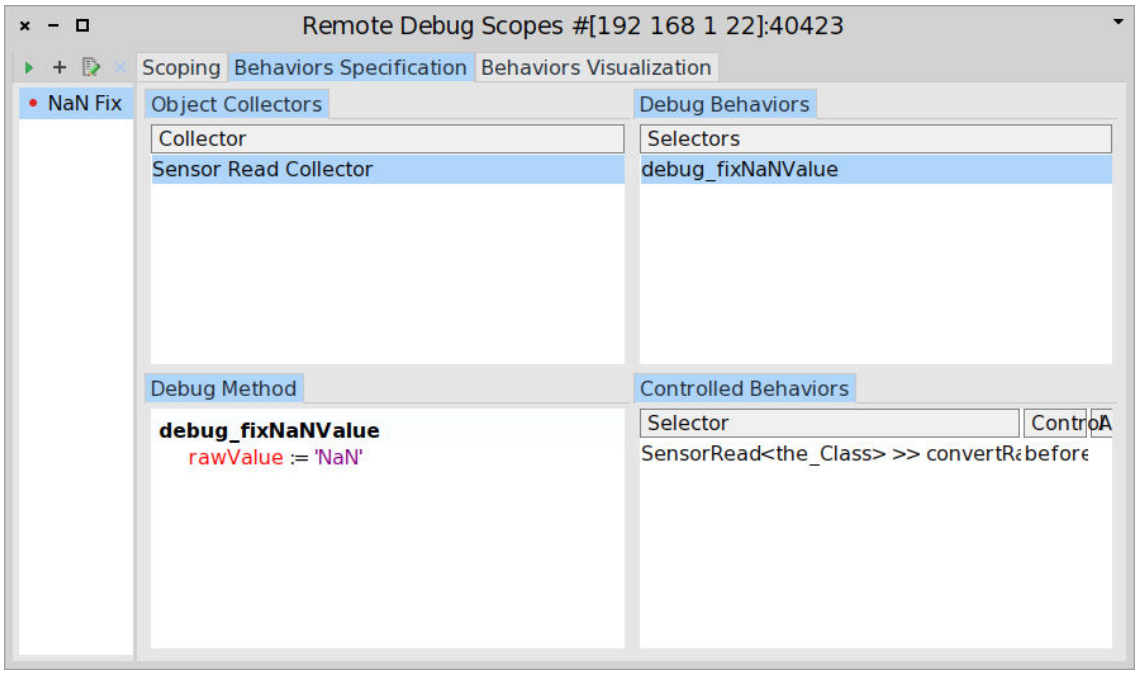


FIGURE 6.6 – La spécification d'adaptation pour un collecteur. Chaque adaptation contient un ensemble de méthodes définies par l'utilisateur-rice ainsi qu'un contrôle, qui spécifie comment une méthode adaptée s'insère dans le comportement d'un objet. Chaque collecteur possède sa propre adaptation, définie par l'ensemble des méthodes et des contrôles définis dans cette interface.

Controlled Behaviors. Le contrôle de comportement spécifie comment une méthode adaptée altère le comportement d'un objet. La Figure 6.7 illustre la définition d'un contrôle à partir du menu contextuel du navigateur de la Figure 6.6. La Figure 6.8 montre l'interface de définition du contrôle de comportement, pour une méthode d'adaptation sélectionnée.

La première colonne (A) présente la liste des classes connues par le collecteur pour lequel l'adaptation est définie. Ces classes correspondent aux classes des objets déjà collectés au moment de la définition de l'adaptation. La seconde colonne (B) présente les méthodes pour la classe sélectionnée. La méthode qui sera sélectionnée sera la méthode qui sera contrôlée par l'adaptation. Dans le cas où la méthode d'adaptation prendrait en entrée des paramètres, comme ici la méthode *debug_ sendPhoto:in:debugging:*, il est nécessaire de spécifier les objets qui seront injectés en tant que paramètres à l'exécution (colonne C). Ces paramètres peuvent être, dans n'importe quel ordre, une réification du contexte d'exécution (Figure 6.9) ou un paramètre de la méthode contrôlée. C'est le cas dans la Figure 6.8 (colonne C), où les paramètres qui seront passés à la méthode d'adaptation sont deux des paramètres de la méthode contrôlée (*server* et *channel*) et une réification qui correspond au nom de la méthode contrôlée.

Plusieurs méthodes différentes peuvent être contrôlées par une même adaptation. Par exemple, une méthode d'adaptation qui génère des traces d'exécution peut s'exécuter avant ou après plusieurs méthodes différentes de l'objet adapté. Dans ce cas, les paramètres injectés dans la méthode d'adaptation peuvent être différents suivant les méthodes contrôlées, car ils dépendent du contrôle défini pour chaque méthode adaptée dans l'interface de la Figure 6.8.

Dans cette mise en œuvre, il est possible qu'un nouvel objet collecté ne soit instance d'aucune des classes contenant des méthodes pour lesquelles une adaptation a été définie. Dans ce cas, cet objet ne sera pas soumis à l'adaptation car aucune spécification n'existe pour ce type d'objet. Il s'agit d'une limitation de cette mise en œuvre, et notamment du choix retenu pour la définition et la visualisation d'adaptation au travers de l'outillage développé.

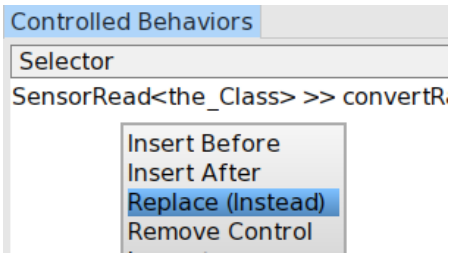


FIGURE 6.7 – Menu contextuel pour la définition d'un contrôle d'adaptation, pour la méthode d'adaptation sélectionnée.

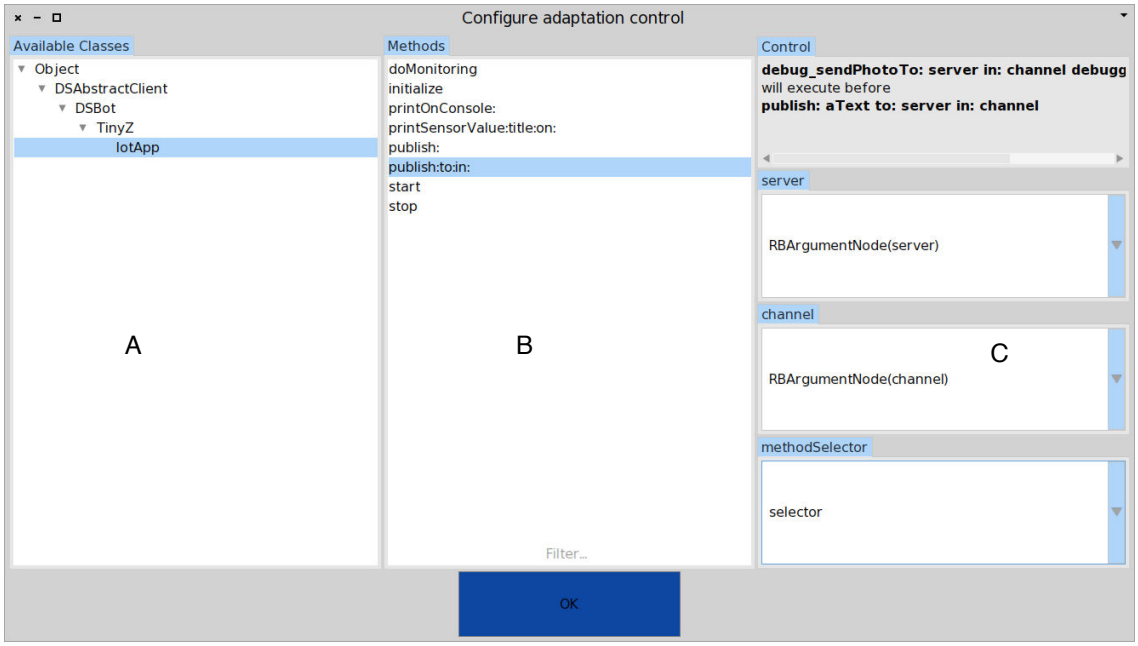


FIGURE 6.8 – Définition du contrôle d'une méthode adaptée. Le contrôle spécifie comment le comportement adapté s'insère dans celui des objets collectés.

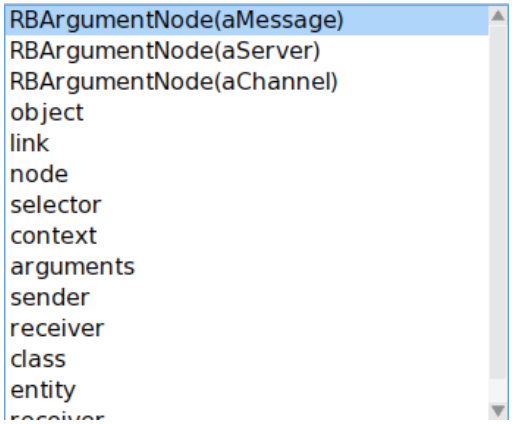


FIGURE 6.9 – Réifications du contexte d'exécution d'une méthode

6.1.4 Visualisation du comportement adapté

Le troisième et dernier onglet du navigateur de périmètres (Figure 6.10) présente une visualisation du comportement qui sera adapté lors de l'activation du périmètre. Il s'agit d'une aide pour l'utilisateur-rice, afin de visualiser globalement la variation comportementale définie par le périmètre. Cette interface montre la liste des classes des objets collectés (*Altered Behaviors for Collected Objects*, première ligne, première colonne), et pour lesquelles des adaptation ont été définies (*Adapted Methods*, première ligne, seconde colonne). Lorsqu'une méthode d'adaptation est sélectionnée pour une classe, la seconde ligne montre respectivement la méthode d'origine (à gauche) — qui correspond à une méthode contrôlée par l'adaptation définie via l'interface de la Figure 6.8 — ainsi que la variation comportementale appliquée par l'adaptation (à droite). Cette interface permet de visualiser concrètement le contrôle appliqué par l'adaptation sur la méthode d'origine, afin de fournir une variation comportementale. Ici, l'adaptation exécutera une nouvelle méthode en préambule de la méthode d'origine. Il est important de préciser que ce ne sont pas les classes listées dans la première colonne qui seront adaptées, mais les objets instances de ces classes qui seront collectés par le collecteur du périmètre.

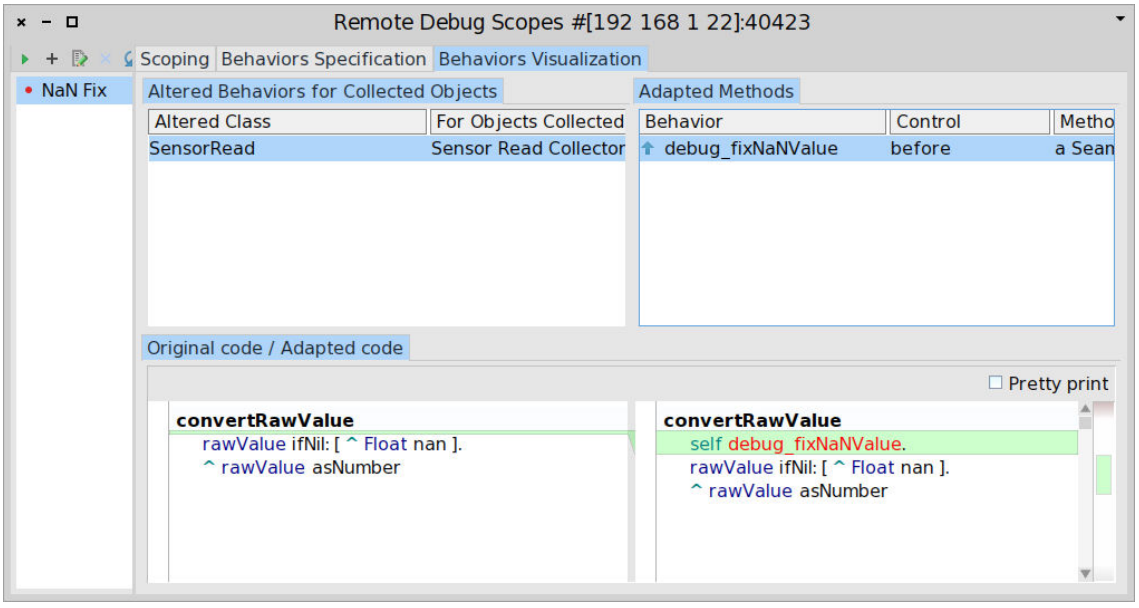


FIGURE 6.10 – Visualisation du comportement adapté pour un périmètre d'adaptation donné. L'impact d'une adaptation est visualisable pour chaque classe dont une instance a été collectée. Pour une adaptation, le code d'une méthode adaptée est affiché côte à côte avec sa version adaptée (ligne du bas).

6.1.5 Compatibilité avec les outils natifs de Pharo

Les variations comportementales définies par l'utilisateur-rice ne sont pas soumises à validation avant application de l'adaptation. Si des erreurs sont présentes dans ces comportements, des exceptions seront soulevées à l'exécution. L'adaptation est cependant intégrée au langage, et reste dans le paradigme objet. Si des exceptions dues au code adapté sont soulevées pendant l'exécution, elles sont traitées comme les exceptions du programme hors du champ de l'adaptation. Une exception est toujours soulevée à partir d'une méthode d'une classe, exécutée sur une instance de cette classe. Notamment, si l'environnement d'exécution du programme contient un déverminer (local ou distant), ce dernier interceptera les exceptions dues à l'adaptation comme les exceptions dues au code original du programme. Il est alors possible de corriger le code de l'adaptation dans le déverminer, et d'effectuer toutes les opérations classiques fournies par l'environnement : exécution pas-à-pas, modification et redémarrage d'une méthode, reprise de l'exécution à partir d'un point précédent de la pile d'exécution...

Il n'y a cependant aucune garantie que l'erreur soit corrigeable, ni que l'erreur ne soit pas fatale au programme et provoque son arrêt. En outre, lorsque le déverminer est ouvert, l'exécution du programme est mise en pause. Cela peut ne pas être acceptable s'il y a une contrainte forte sur la continuité d'exécution du programme, et que ce dernier ne doit jamais s'arrêter.

6.2 Cas d'utilisation sur un objet connecté : l'application *Sensor Monitoring App* et le *chat-bot IotZ*

L'application *Sensor Monitoring App* est un cas d'utilisation développé pendant cette thèse [Marra et al., 2017, Costiou et al., 2018b]. Il s'agit d'une application connectée mono-processus développée avec Pharo et déployée sur une carte Raspberry-pi [1]. L'ensemble forme un objet connecté (Figure 6.11) qui possède un capteur de lumière (1), un capteur de température (2) ainsi qu'une caméra (3). L'application est complétée par un *chat-bot* nommé *IotZ*, basé sur un chat-bot Pharo open-source [2]. *IotZ* est entièrement intégré à l'application *Sensor Monitoring App*. *IotZ* fournit un service de surveillance de la luminosité et de la température d'une pièce à des utilisateurs distants, ainsi qu'un service d'interaction avec l'application au travers d'une interface textuelle. Pour ce faire, le bot s'interface avec un serveur de discussion Discord [3].

1. https ://www.raspberrypi.org/products/raspberry-pi-3-model-b/
2. https ://github.com/ClotildeToullec/TinyZ
3. https ://discordapp.com/

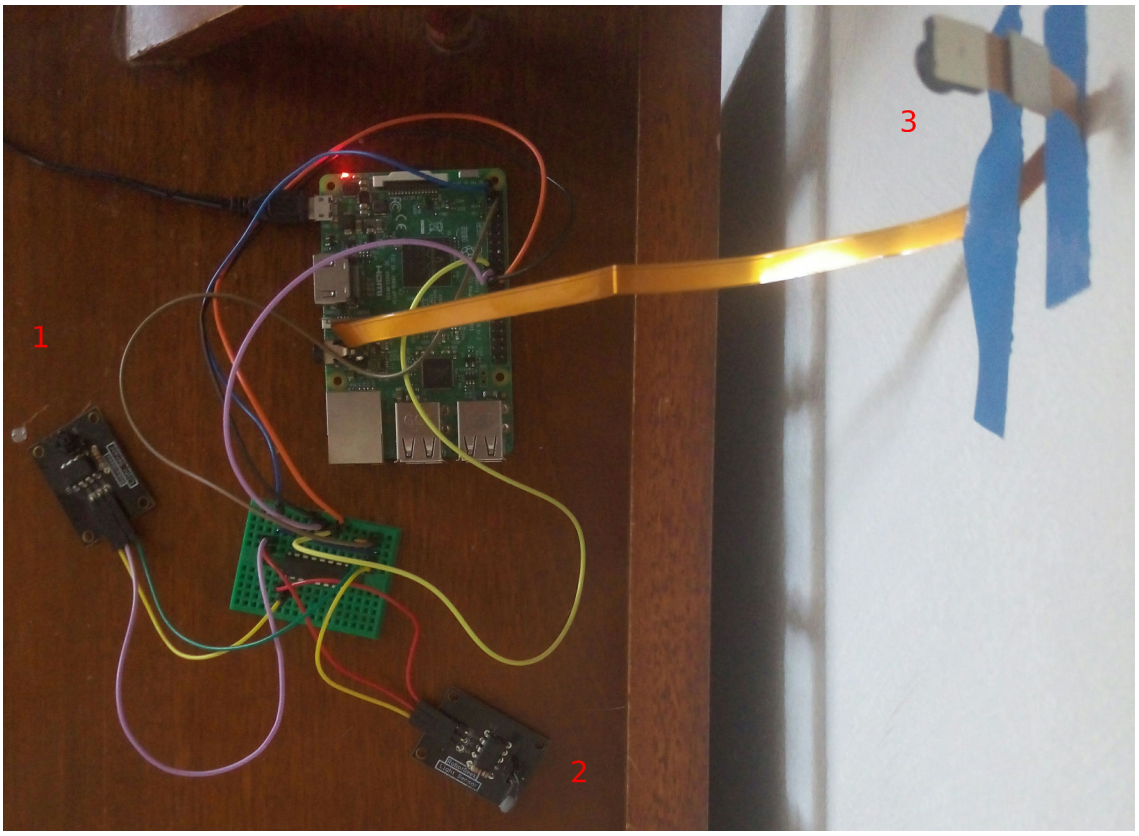


FIGURE 6.11 – L'application connectée *Sensor Monitoring App* en fonctionnement. (1) capteur de lumière. (2) capteur de température. (3) caméra.

Toutes les dix secondes, *IotZ* effectue une lecture de capteur, transforme les données brutes et les transmet au serveur de discussion. La Figure 6.12 montre un extrait des données transmises au serveur, et visibles par tous les clients. Chaque client peut communiquer avec le serveur au travers d'une interface textuelle. Lorsqu'un client utilise cette interface, le bot interprète la commande et si cette dernière est valide, répond dans le salon de discussion. Par exemple lorsqu'un client tape la commande *z help*, le bot envoie son manuel d'utilisation sur le salon de discussion. La caméra n'est pas active par défaut, bien que l'application fournisse dans son code source des fonctionnalités de capture d'image.

L'application est soumise à des contraintes, énumérées ci-après, qui illustrent des cas d'utilisation concrets rencontrés dans des applications en production.

— Le service ne doit pas être interrompu. Dans le cas d'un problème, il n'est pas souhaitable de déverminer le programme en y plaçant des points d'arrêt. Il n'est pas non plus souhaitable d'arrêter l'application pour y insérer du comportement spécifique à l'activité de déverminage (par exemple des traces).

— Certaines erreurs n'apparaissent que sur le lieu de déploiement de l'objet connecté. Lorsque l'appareil est ramené en environnement de développement (ou à l'usine) il est très difficile de reproduire les erreurs, car elles dépendent du contexte dans lequel est déployé l'appareil. Le déverminage doit donc idéalement s'effectuer sur place.

— Dans le cas d'un problème, ou d'une donnée suspecte, les utilisateur-rice-s expert-e-s du domaine sont les plus à même d'interpréter la donnée et de décider si elle est problématique. Un-e développeur-euse ne peut pas décider seul-e de la pertinence d'un résultat et de modifier l'application en conséquence.

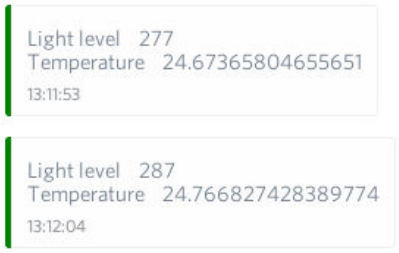


FIGURE 6.12 – Extrait de données de surveillance affichées dans la discussion.

La Figure 6.13 illustre les conditions expérimentales du cas d'utilisation étudié. Des utilisateur-rice-s expert-e-s, avec une connaissance du domaine et du contexte des données de surveillance, sont connecté-e-s via différents clients au serveur de discussion. Un-e développeur-euse est connecté-e à l'application avec l'outillage distant des périmètres d'adaptation. Dans les expérimentations qui suivent, nous jouons simultanément le rôle d'un-e utilisateur-rice expert-e et d'un-e développeur-euse.

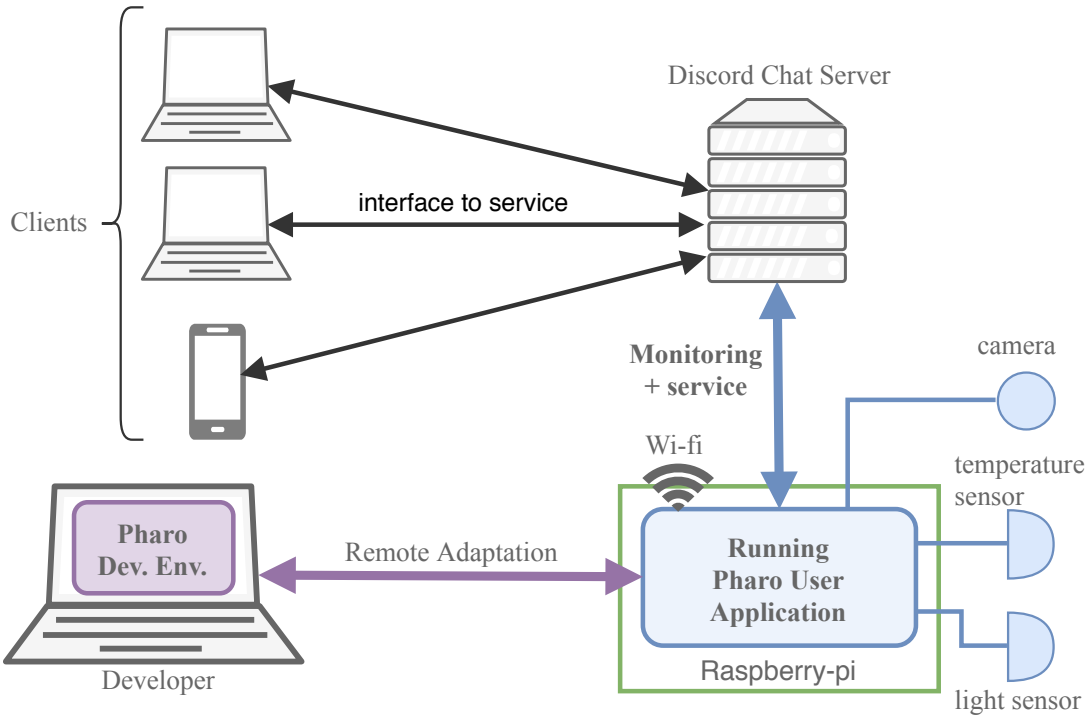


FIGURE 6.13 – L'application connectée *Sensor Monitoring App* avec son *chat-bot IotZ*. Le bot utilise les fonctionnalités de l'application pour fournir des données de surveillance environnementale ainsi que des services sur un serveur de discussion. Des utilisateur-rice-s expert-e-s visualisent les données de surveillance et interagissent avec le bot. Un-e développeur-euse est connecté-e à distance au programme, avec l'environnement et l'outillage des périmètres d'adaptation.

6.3 Scénario 1 : déverminage non-anticipé en cours d'exécution

Dans ce scénario, une exception provoque l'envoi de données non-interprétables pour la surveillance de l'environnement de l'objet connecté. Ces données provoquent en outre une pollution du serveur de discussion. Ce problème est visible dans l'extrait de données du serveur de discussion de la Figure 6.14. Ce problème est imprévisible et se produit aléatoirement, mais uniquement dans l'environnement de production. Il n'est pas reproductible en environnement de développement, et tous les tests unitaires de l'application déployée se terminent avec succès. Avec les périmètres d'adaptation, nous proposons d'identifier les objets qui produisent l'erreur puis de la corriger sous la forme d'une adaptation non-anticipée de comportement.

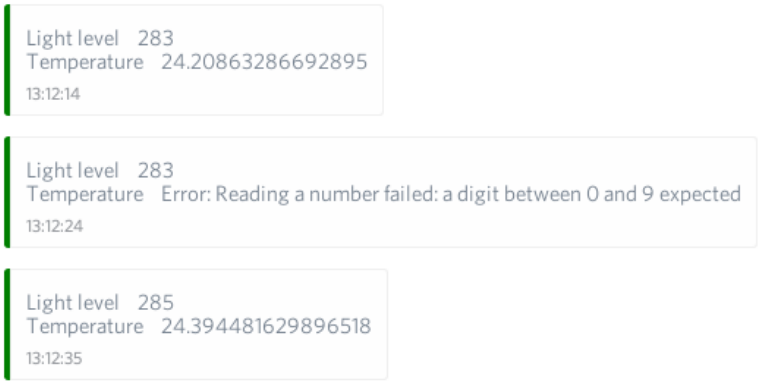


FIGURE 6.14 – Extrait de données de surveillance problématiques affichées sur le serveur de discussion.

6.3.1 Détection du problème

Les données problématiques concernent uniquement le capteur de température. Parfois, il semble y avoir une erreur de lecture d'un nombre – c'est la seule information visible sur le serveur de discussion. L'inspection du code montre que les valeurs brutes lues sur les capteurs sont encapsulées dans des objets temporaires, instances d'une classe *SensorRead*. Les méthodes de lecture sont au nombre de trois, et sont illustrées dans le listing ci-dessous. Le caractère ^ correspond à l'instruction de retour d'une méthode, ou *return* dans des langages comme Java.

```
1  read: address
2      channel := address.
3      self basicRead: address.
4      value := self convertRawValue
5
6  basicRead: address
7      "Lecture du capteur dans un bloc de gestion d'erreur"

8      rawValue := [ bus read: channel ] "lecture physique..."
9          on: SensorReadError      "...capture d'erreurs..."
10         do: [ 'nan' ]            "...traitement de l'erreur."
11
12 convertRawValue
13     rawValue ifNil: [ ^ Float nan ].
14     ^rawValue asNumber
```

Lors d'une lecture, les objets *SensorRead* font appel à la méthode *read:* (ligne 1). Une méthode de lecture bas niveau est d'abord appelée (*basicRead:*, ligne 3).

Cette dernière stocke le résultat brut de la lecture dans une variable $rawValue$ (ligne 7). Il semble que des erreurs de lectures puissent apparaître (ligne 8) et dans un tel cas la chaîne de caractères *'nan'* est renvoyée comme résultat de lecture (ligne 10). De temps à autre, la lecture physique sur un capteur peut effectivement échouer, et soulève des exceptions du type $SensorReadError$. Ce cas est traité en forçant la valeur qui sera stockée dans la variable $rawValue$ à la chaîne de caractères *'nan'*. Une méthode de conversion est ensuite appelée (ligne 4). Cette méthode, $convertRawValue$ (ligne 11), vérifie d'abord si la valeur est nulle, avant de la convertir avec le message $asNumber$ (ligne 13). Quand la valeur brute est invalide, la conversion doit donc renvoyer le flottant spécial *NaN*, qui signifie *Not a Number*.

Dans Pharo, la chaîne de caractères *'nan'* en minuscules ne peut pas être convertie en flottant, car le code de conversion est sensible à la casse. En revanche, la chaîne de caractères *'NaN'* peut répondre au message $asNumber$ et renvoie le flottant spécial NaN. Il y a donc au moins une erreur de frappe dans le code de la méthode *basicRead*: – les majuscules ont été oubliées – et une erreur dans un test unitaire puisque le problème n'a pas été détecté avant le déploiement.

6.3.2 Correction du problème par une adaptation de comportement

Nous proposons d'adapter dynamiquement les instances de $SensorRead$ pour lesquelles une valeur brute erronée serait retournée suite à une erreur de lecture. Nous venons de formuler l'hypothèse qu'il s'agit d'une erreur de frappe (*'nan'* au lieu de *'NaN'*). L'adaptation nous permet donc d'expérimenter une correction et de valider notre hypothèse, et éventuellement de rester en place pour corriger le problème jusqu'à la prochaine mise à jour du programme. Celui-ci doit en effet continuer à fournir son service sans interruption. Les différentes étapes de l'adaptation sont les suivantes, résumées par les Figures 6.15 et 6.16 :

1. **Périmètre**. Nous définissons un nouveau périmètre d'adaptation vide, nommé *NaN Fix*. Toutes les opérations suivantes concernent ce périmètre.

2. **Collecteur**. Dans la méthode *read*: de la classe $SensorRead$, nous définissons un point de collecte sur le receveur du message *basicRead*:. À l'exécution, l'objet référencé par $self$ sera collecté, c'est-à-dire l'instance de $SensorRead$ exécutant la méthode *read*:. Le receveur est collecté **avant** l'envoi du message *basicRead*:. Lorsque la méthode correspondante s'exécute, l'objet $self$ est donc déjà collecté. La Figure 6.1 illustre la définition du collecteur, qui est résumée dans le montage de la Figure 6.15.

3. **Adaptation**. Nous définissons une adaptation de comportement. Il s'agit d'une méthode nommée $fixNaNValue$, qui force la valeur brute à la chaîne de caractères *'NaN'* dans la variable d'instance $rawValue$ de l'objet adapté. La Figure 6.6 illustre la définition de l'adaptation, qui est résumée dans le montage de la Figure 6.16.

4. **Contrôle**. La méthode d'adaptation tout juste définie doit maintenant s'insérer dans le comportement de l'instance de $SensorRead$ collectée. Nous définissons un contrôle d'adaptation pour exécuter cette méthode avant la méthode $convertRawValue$. La mise en place du contrôle est illustrée par les Figures 6.7 et 6.8, et résumée dans le montage de la Figure 6.16.

5. **Activateur**. Si le périmètre est activé, l'adaptation sera appliquée sans condition à toutes les instances de $SensorRead$ collectées. La valeur de la variable d'instance $rawValue$ sera donc systématiquement forcée à *'NaN'*. Cette modification comportementale n'a de sens que si la valeur stockée dans $rawValue$ est erronée suite à une lecture de capteur en échec. Nous définissons un activateur − c'est-à-dire une garde d'adaptation − sur la variable $rawValue$ de la méthode *basicRead:*. L'activateur cible précisément le point du flot de contrôle qui correspond à l'affectation d'une nouvelle valeur dans cette variable, dans la méthode *basicRead:*. À chaque nouvelle affectation, une condition sera évaluée, et si elle est évaluée au littéral *true* le périmètre d'adaptation sera activé. Ici il s'agit de vérifier si la chaîne de caractères problématique *'nan'* est stockée dans $rawValue$ suite à une lecture sur un capteur. Tout autre affectation à $rawValue$ hors de ce flot de contrôle n'est pas instrumentée par l'activateur. La définition de l'activateur est illustrée par la Figure 6.1, et résumée dans le montage de la Figure 6.15.

Le flot de contrôle des méthodes illustrées à la section 6.3.1 est impacté comme suit :

1. Lorsqu'une instance de la classe $SensorRead$ exécute la méthode *read:*, cette instance est collectée ($self$) avant de recevoir le message *basicRead:*.
2. Lorsque la méthode *basicRead:* est exécutée, si la valeur retournée par la lecture de capteur vaut *'nan'*, le périmètre d'adaptation est activé et les objets collectés (ici uniquement $self$) sont adaptés.
3. Lorsque la méthode $convertRawValue$ est exécutée et si l'objet courant a été adapté, la méthode *debug_fixNaNValue* est exécutée en préambule et force la valeur de la variable $rawValue$ à *'NaN'*.

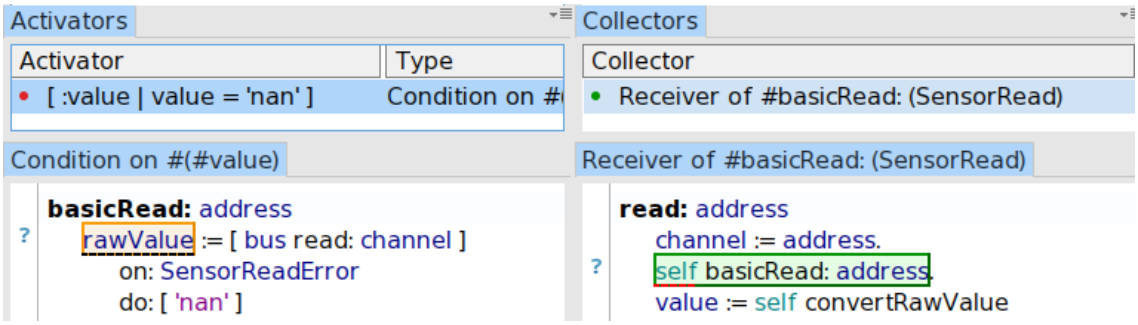


FIGURE 6.15 – Montage de la Figure 6.1. Colonne de gauche : la garde d'adaptation (en haut) qui sera évaluée à chaque fois qu'une valeur sera assignée dans la variable d'instance $rawValue$ d'un objet de la classe $SensorRead$ (en bas). Si après l'affectation dans ce flot de contrôle précis, $rawValue$ vaut *'nan'*, le périmètre est activé. Colonne de droite : collecteur sur le receveur ($self$) du message *basicRead:*, dans la méthode *read:* de la classe $SensorRead$. La collecte est effectuée avant la réception du message *basicRead:* par $self$. Les expressions instrumentées sont annotées visuellement par l'outil (surlignage et icône en marge).

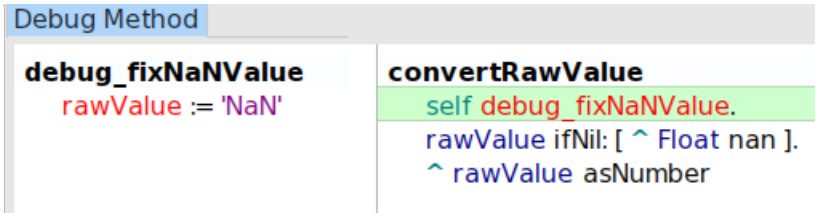


FIGURE 6.16 – Adaptation de comportement appliquée à la méthode $convertRawValue$ des instances collectées de la classe $SensorRead$. Montage des Figures 6.6 et 6.10.

Le résultat de l'adaptation sur le serveur de discussion est visible à la Figure 6.17. La valeur représentant le flottant NaN est affichée lorsque le périmètre d'adaptation s'active. Le résultat est désormais compréhensible par les utilisateur-rice-s comme un échec de lecture sur un capteur. L'adaptation peut ensuite rester en place, jusqu'à une prochaine mise à jour du programme qui intégrera un correctif.

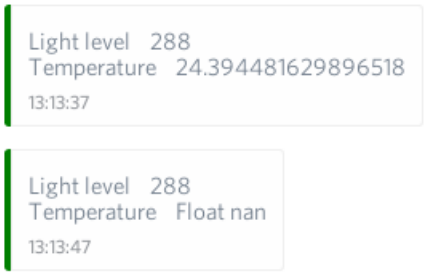


FIGURE 6.17 – Lorsqu'une lecture de capteur provoque une erreur, le périmètre *NaN Fix* est activé et les objets *SensorRead* collectés sont adaptés. La valeur envoyée au serveur de discussion correspond au flottant spécial NaN et n'affiche plus d'erreur.

6.3.3 Discussion

Dans cette section, nous discutons d'aspects relatifs à ce scénario. Une discussion générale fait un bilan plus global des périmètres d'adaptation à section 6.6.

Ce scénario illustre le cas d'une application déployée dans un environnement dont le contexte provoque un comportement imprévu. La lecture problématique du capteur a été anticipée et prévue dans le code du programme, mais son traitement comporte une erreur qui est passée au travers des tests unitaires. Il s'agit d'un *bug*, problème par définition non-anticipé, et qui s'avère problématique lorsque le logiciel est déployé et ne doit pas être interrompu. L'adaptation des objets permet de corriger le problème, en tous cas d'éviter une pollution du serveur de discussion par des traces d'erreurs sans signification pour l'utilisateur-rice final-e (de son point de vue, il s'agit bien d'une correction).

Nous proposons au travers des *Debug-Scopes* une manière d'expérimenter des modifications comportementales du programme en cours d'exécution. L'expérimentation permet ici de valider une hypothèse concernant la source du problème − l'erreur de casse sur la chaîne de caractères *'nan'* − et d'expérimenter une correction. L'expérimentation sur le programme en cours d'exécution est importante pour valider un correctif. Il n'y a en effet pas de garanties de pouvoir reproduire l'erreur si l'objet connecté redémarre suite à une mise à jour, ou s'il est ramené en usine.

Cet exemple ne met pas l'accent sur le grain de l'adaptation. Bien que le grain se situe au niveau objet − des instances spécifiques de la classe $SensorRead$ sont bien adaptées indépendamment les unes des autres − le cloisonnement des objets adaptés est ici lié au flot de contrôle du programme. Comme le programme est séquentiel, la garde d'adaptation est évaluée avant chaque appel à la méthode $convertRawValue$ pour chacune des instances collectées de $SensorRead$. Autrement dit, toutes les instances de $SensorRead$ sont collectées car le point de collecte se situe sur une

expression évaluée par toutes les instances de cette classe. Puis, chaque fois que la garde d'adaptation est évaluée à *true* dans le flot de contrôle, l'adaptation est appliquée aux instances collectées et toujours présentes en mémoire. Lorsque la garde est évaluée à $false$, l'adaptation est désactivée. Une garde de collecte aurait été plus pertinente pour filtrer les objets. Seuls les objets filtrés pourraient alors être soumis à l'adaptation, du moment que les gardes d'adaptation évaluent à *true* (les activateurs). Cette méthode est plus indiquée dans le cas d'objets persistants en dehors du flot de contrôle instrumenté, et dont le périmètre de l'adaptation doit être plus précisément cloisonné pour éviter les effets de bord.

6.4 Scénario 2 : utilisation non-anticipée de la caméra pour la prise de photographies de surveillance

Dans ce scénario, une montée soudaine et inexplicable de la température est visible de temps à autre et de manière imprévisible sur le serveur de discussion. À cause du délai entre le moment de la mesure et la réception des données sur le serveur de discussion, lorsqu'un-e utilisateur-rice est averti-e du problème la température peut déjà être redescendue. Si ça n'est pas le cas, le temps de déplacement jusqu'à l'objet connecté pour vérifier ses conditions environnementales suffit pour que la température redescende à un niveau stable. Il est donc difficile de comprendre la raison de cette montée de température. Les utilisateur-rice-s expert-e-s du domaine souhaitent alors visualiser l'environnement direct de l'objet connecté. L'objectif est de produire une trace dédiée sur le serveur de discussion sous la forme d'une photographie, lorsqu'une poussée de température a lieu. Cette photographie peut être effectuée en utilisant la caméra de l'objet connecté, qui est inutilisée par défaut dans cette application.

6.4.1 Adaptation du programme pour utiliser la caméra

Nous définissons d'abord un nouveau périmètre d'adaptation nommé *Advanced Logging*. Toutes les opérations suivantes concernent ce périmètre. Un point de collecte est défini dans la méthode $doMonitoring$ de la classe $IotApp$ pour collecter l'instance unique du bot en cours d'exécution (Figure 6.18). Le programme passe cycliquement dans cette méthode, mais la collecte n'aura lieu qu'une seule fois pour le même objet (le bot). Cette collecte a lieu juste avant la publication des données issues des capteurs sur le serveur de discussion. Cette publication est mise en œuvre dans la méthode *publish*: de la même classe.

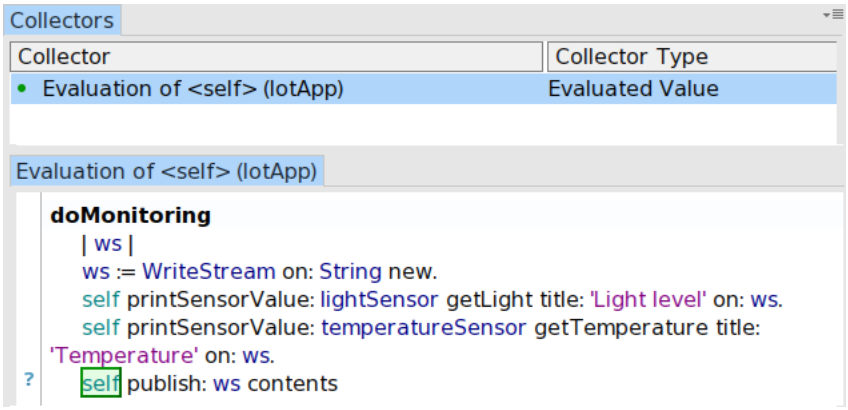


FIGURE 6.18 – Définition d'un point de collecte sur *self* de la méthode *doMonitoring* de la classe *IotApp*. L'instance de cette classe qui sera collectée correspond au *chat-bot* actif.

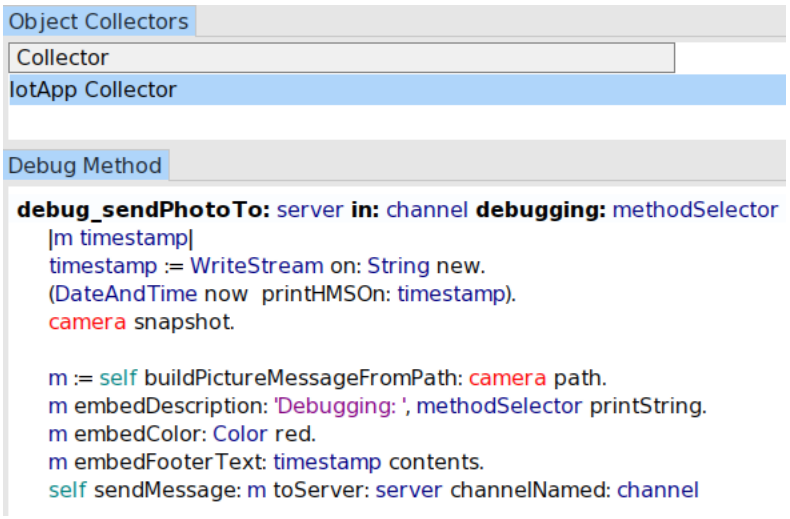


FIGURE 6.19 – Définition d'une adaptation qui utilise la caméra du *bot* pour prendre une photo et l'envoyer sur le serveur de discussion.

Nous définissons ensuite une adaptation de comportement qui utilise la caméra pour prendre une photo, puis l'envoyer sur le serveur de discussion (Figure 6.19). Nous utilisons la variable d'instance *camera* de l'objet collecté, qui contient une interface permettant de prendre des photos. Cette méthode d'adaptation attend trois paramètres à l'exécution : le nom du serveur de discussion, le canal de publication et le nom de la méthode qui est instrumentée.

```
1       publish: aText
2           self publish: aText to: 'TheServer' in: 'iot-experiments'

3
4       publish: aText to: server in: channel
5           [...publication code...]
```

FIGURE 6.20 – Méthodes de publication sur le serveur de discussion (classe *IotApp*).

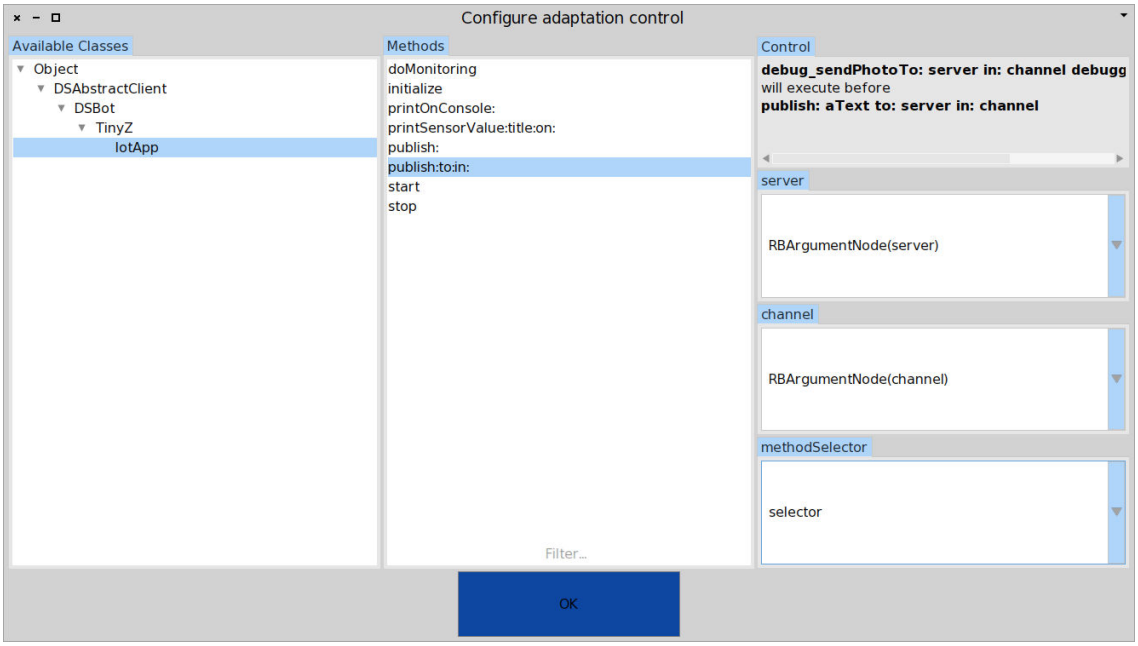


FIGURE 6.21 – L'adaptation de comportement est configurée pour être exécutée avant la méthode *publish:to:in:* de la classe *IotApp*, et prendra en paramètres deux arguments de cette méthode et une réification du contexte d'exécution correspondant au nom de la méthode exécutée.

Nous décidons d'insérer ce comportement au moment de la publication des données de surveillance sur le serveur de discussion. Cette publication a lieu au travers de la méthode *publish:* (Figure 6.20). Elle fait appel à une autre méthode *publish:to:in:* qui prend en paramètres le texte à publier, le nom du serveur et le nom du canal sur lequel envoyer le message. Nous décidons d'adapter cette dernière méthode en insérant la méthode d'adaptation juste avant son exécution.

Le contrôle de l'adaptation est illustré par la Figure 6.21. La méthode d'adaptation définie précédemment (Figure 6.19) est insérée avant la méthode *publish:to:in:* de la classe *IotApp*. Les arguments sélectionnés pour être injectés en paramètres à la méthode d'adaptation sont d'abord les arguments *server* et *channel* de la méthode *publish:to:in:*, puis une réification du contexte qui correspond au sélecteur de la méthode en cours d'exécution. À noter que l'ordre des paramètres des deux méthodes est différent. Par exemple, l'argument *server* est le premier paramètre passé à la méthode d'adaptation, alors qu'il s'agit du second paramètre passé à la méthode *publish:to:in:*. Il n'y a pas de contraintes sur l'ordre des paramètres passés aux méthodes d'adaptation lors de la définition du contrôle et lors de leur exécution. La Figure 6.22 montre la visualisation de l'adaptation de la méthode *publish:to:in:* des instances de la classe *IotApp* qui seront adaptées.

```
publish: aText to: server in: channel
    | m timestamp |
    timestamp := WriteStream on: String new.
    DateAndTime now printHMSOn: timestamp.
    m := self message.
    m embedDescription: aText.
    m embedColor: Color green muchDarker.
    m embedFooterText: timestamp contents.
    self sendMessage: m toServer: server channelNamed: channel

publish: aText to: server in: channel
    | timestamp m |
    self debug_sendPhotoTo: server in: channel debugging: #selector.
    timestamp := WriteStream on: String new.
    DateAndTime now printHMSOn: timestamp.
    m := self message.
    m embedDescription: aText.
    m embedColor: Color green muchDarker.
    m embedFooterText: timestamp contents.
    self sendMessage: m toServer: server channelNamed: channel
```

FIGURE 6.22 – La méthode d'adaptation *debug_sendPhotoTo:in:debugging:* s'exécutera juste avant la méthode *publish:to:in:* des instances de la classe *IotApp* lorsque celles-ci seront adaptées.

Nous devons maintenant définir les conditions d'activation du périmètre d'adaptation. Deux activateurs sont spécifiés, et sont visibles dans les Figures 6.23 et 6.24.

Activateur conditionnel. La demande utilisateur concerne la production de traces — sous la forme de photographie — uniquement lorsque la température dépasse les 30^o Celsius. Un activateur conditionnel est donc placé sur l'instruction

qui récupère la température dans la méthode *doMonitoring* (Figure 6.23). Lorsque l'expression sélectionnée est évaluée, l'activateur exécute sa condition avec la valeur issue de cette évaluation. Si le résultat est *true*, une demande d'activation est envoyée au périmètre, sinon le périmètre est désactivé.

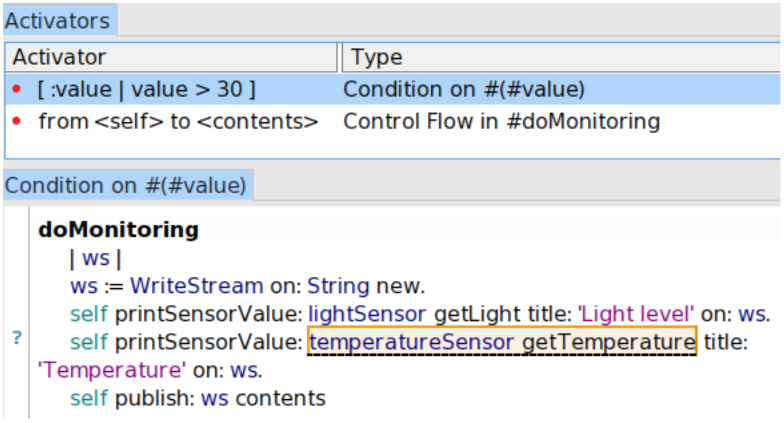


FIGURE 6.23 – Garde d'adaptation sous forme d'activateur conditionnel

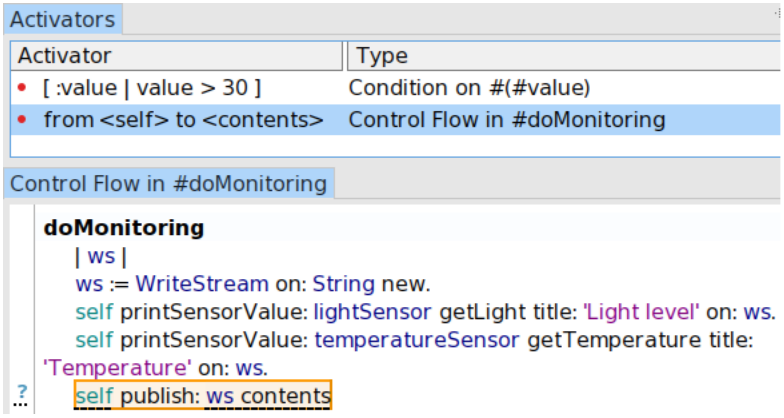


FIGURE 6.24 – Garde d'adaptation sous forme d'activateur basé sur le flot de contrôle

Activateur basé sur le flot de contrôle. Si la condition de l'activateur conditionnel est évaluée à *true*, le périmètre applique l'adaptation sur les objets collectés. Ici il s'agit de l'unique instance collectée de la classe *IotApp*. Si le périmètre était activé à ce moment, il ne pourrait pas être désactivé avant que

l'activateur n'évalue à nouveau sa condition et que cette dernière ne soit évaluée à $false$. Autrement dit, il faut attendre que le programme repasse par le flot de contrôle de la méthode *doMonitoring* où est défini l'activateur. Si la méthode adaptée *publish:to:in:* est appelée hors de ce contexte, l'adaptation sera toujours active et enverra également une photo. Nous souhaitons que l'adaptation soit active uniquement lorsque le programme passe dans la boucle de surveillance, c'est-à-dire la méthode *doMonitoring*. Nous définissons un activateur basé sur le flot de contrôle de cette méthode (Figure 6.24). Ce dernier est configuré pour envoyer une demande d'activation au périmètre quand le programme traverse le flot de contrôle sélectionné (en orange). En dehors de ce flot de contrôle, le périmètre et la variation comportementale qu'il décrit sont inactifs.

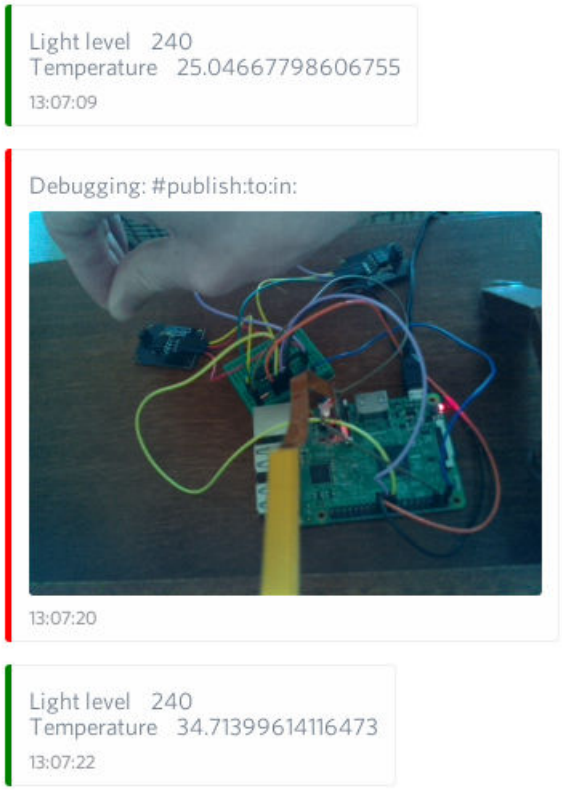


FIGURE 6.25 – La trace sous forme de photographie sur le serveur de discussion, lorsque les conditions d'adaptation sont réunies, permet de visualiser la raison de la montée de température.

Le résultat de l'adaptation est visible à la Figure 6.25. Le bot publie une photo du dispositif grâce à l'adaptation comportementale qui utilise la caméra. L'adaptation est activée car le programme est passé dans le flot de contrôle de la publication des résultats sur le serveur de discussion et que la température mesurée est supérieure à 30^o Celsius. Dans notre cas, la raison de la montée de température peut être directement déduite du contexte environnemental... ici un opérateur a la main sur le capteur. Le périmètre peut être manuellement désactivé pour retirer le comportement de trace, et remis en place à la demande.

6.4.2 Discussion

Dans cette section nous discutons d'aspects relatifs à ce scénario. Une discussion générale fait un bilan plus global des périmètres d'adaptation à la section 6.6.

Ce scénario illustre un cas d'utilisation où des utilisateur-rice-s expert-e-s d'un service souhaitent augmenter les capacités de ce dernier pour comprendre un fonctionnement étrange du système. Il s'agit de produire des visualisations du système pour comprendre son comportement, et éventuellement déceler un problème et trouver sa source. Ce scénario est une reproduction d'un cas non-anticipé de cette expérience, où un acteur imprévu de l'environnement (un chat) est venu jouer avec les capteurs, faisant par moments baisser la luminosité et monter la température avant de s'en aller. La demande de visualisation sous forme de photographie est spécifique, car dans ce cas précis la poussée de température pourrait être difficile à comprendre avec des traces purement textuelles. Sa mise en œuvre par l'adaptation dynamique est non-anticipée, l'utilisation de la caméra étant à l'origine imprévue dans le fonctionnement normal du programme. Ce processus de déverminage est dépendant de la connaissance du système des utilisateur-rice-s, qui demandent des visualisations précises et particulières qui ne sont pas toujours réalisables par le biais de l'adaptation de comportement.

Ce scénario présente plusieurs éléments en faveur de l'adaptation non-anticipée de comportement par le biais des *Debug-Scopes*, comparé au redéploiement d'une version spécifique de l'application avec des traces. Ces éléments sont présentés dans les deux paragraphes suivants.

Exploration dynamique du comportement du programme. S'agissant d'un cas exploratoire où l'on cherche à comprendre le fonctionnement du système, la forme de la visualisation peut être amenée à varier. Par exemple, la condition sous laquelle il faut activer la caméra peut être précisée pour cibler des cas plus spécifiques. L'utilisateur-rice pourrait demander à visualiser des informations textuelles

supplémentaires, et à en supprimer certaines qui ne lui semblent finalement pas pertinentes. Cette façon de procéder exploratoire est lourde à mettre en place avec un processus classique d'arrêt-mise à jour-redémarrage de l'application. L'utilisation de techniques comme la mise à jour à chaud [Seifzadeh et al., 2013] est une alternative à l'adaptation dynamique si elle peut être effectuée de manière non-anticipée. En revanche, la mise à jour à chaud implique une modification du code du programme par l'insertion de traces.

Alternative par insertion de traces. Une alternative à l'adaptation consiste à insérer les traces directement dans le code, soit par un processus de déverminage classique en redémarrant l'application avec un nouveau code, soit en effectuant une mise à jour à chaud du programme. Cette technique peut provoquer des situations non désirables en terme de maintenance, et notamment :

— Il existe un risque d'oublier des traces dans le code en production une fois la session de déverminage terminée.
— La lisibilité du code source est polluée par les traces.
— Si les traces sont insérées durablement, par exemple pour observer le programme sur le long terme pour étudier des problèmes imprévisibles et difficiles à reproduire, cela force à maintenir plusieurs versions de l'application en production déployée chez différents client. Les versions instrumentées doivent également recevoir des mises à jour standards de l'application, ce qui peut poser des problèmes de compatibilité.

Dans ce dernier cas, les périmètres d'adaptation peuvent être invalidés par une mise à jour du programme. Cependant, ils ne perturberont pas la mise à jour et la maintenance du système, qui peuvent s'effectuer en toute indépendance des adaptations mises en place. La spécification des périmètres étant toujours disponible, il est possible de les remettre en place après une telle mise à jour.

6.5 Scénario 3 : extension d'interface utilisateur pour du déverminage

Les utilisateur-rice-s du bot trouvent finalement très intéressante cette possibilité d'obtenir directement un visuel de l'objet connecté sur le serveur de discussion. Ces dernier-e-s souhaitent désormais avoir la possibilité d'obtenir une photographie à la demande, au travers d'une interaction avec le bot. L'objectif est ici d'étendre dynamiquement l'interface de communication avec l'application connectée, afin de fournir une nouvelle capacité de visualisation.

6.5.1 Adaptation de l'interface du *chat-bot*

L'adaptation du périmètre *Advanced Logging* du scénario 2 (section 6.4) est augmentée par des méthodes d'adaptation permettant d'étendre l'interface de communication avec le bot. Une commande d'interface *snapshot* est rajoutée au bot, qui est une instance de la classe *IotApp* déjà collectée par le collecteur du périmètre *Advanced Logging*. La Figure 6.26 montre les deux méthodes rajoutées à l'adaptation.

Ajout du traitement de la nouvelle commande. Le comportement adapté est inséré juste avant la méthode qui analyse et traite les messages venant du serveur de discussion. Cette méthode est visible dans la Figure 6.26. Elle prend en paramètre le message reçu dans la discussion. Si le message reçu correspond à la nouvelle commande, la méthode d'adaptation définie à la section 6.4 est appelée afin de prendre une photographie et de l'envoyer au serveur de discussion.

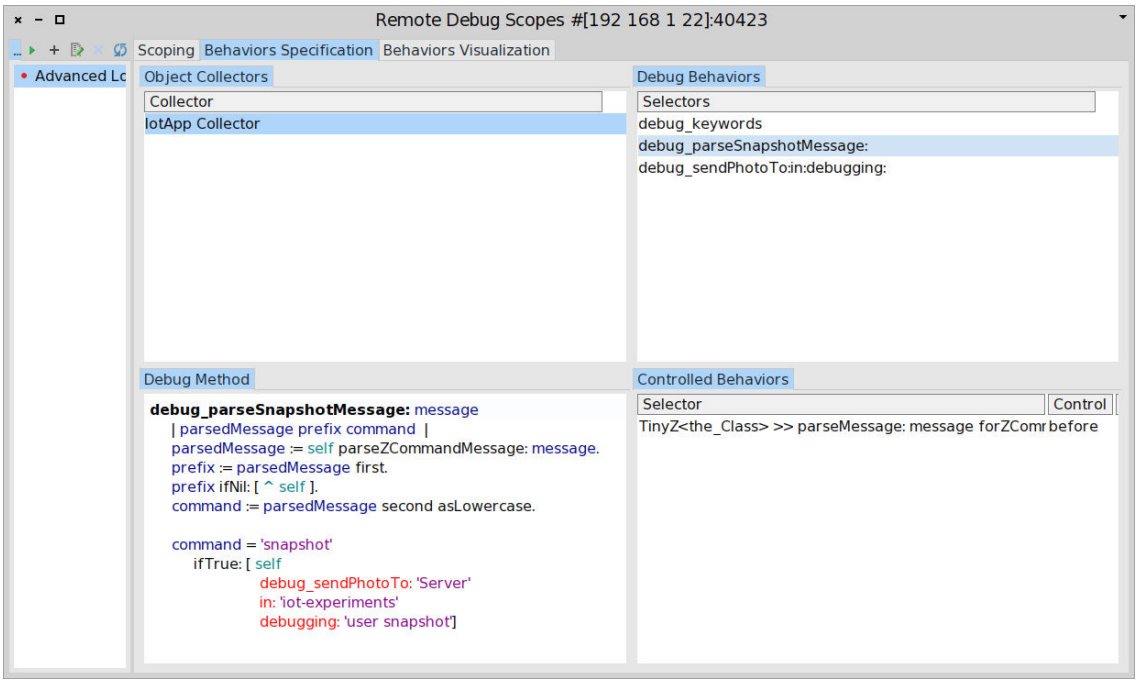


FIGURE 6.26 – Définition de l'adaptation d'extension de l'interface du bot. L'adaptation du périmètre *Advanced Logging* est complétée avec deux méthodes qui adaptent le comportement de l'instance de *IotApp* collectée (le bot). La méthode d'adaptation principale, *debug_parseSnapshotMessage:*, réutilise la méthode d'adaptation définie à la section 6.4 pour utiliser la caméra sous réception de la commande *snapshot*.

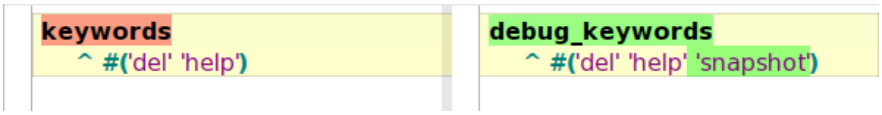


FIGURE 6.27 – Une nouvelle méthode adapte et remplace la méthode *keywords* du bot pour augmenter la liste des commandes valides auxquelles ce dernier peut répondre. Le chapeau ^ représente l'instruction de retour (*return*).

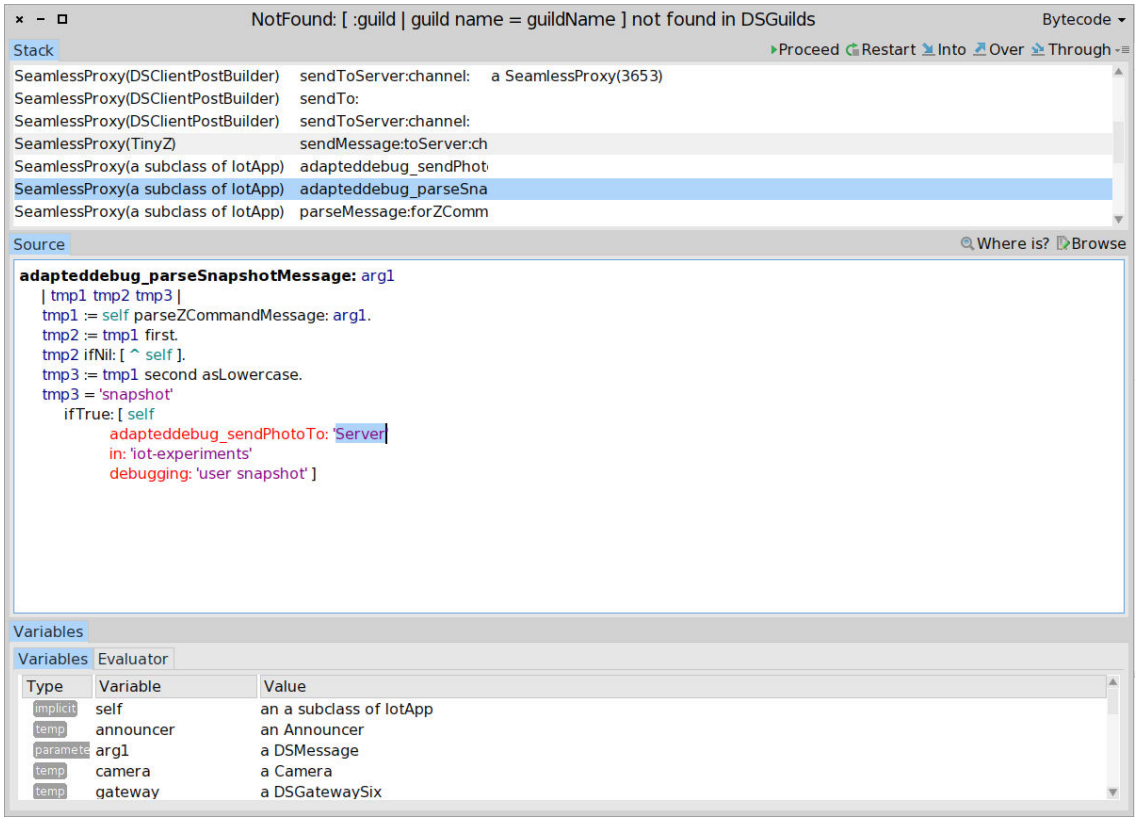


FIGURE 6.28 – Lors d'une erreur dans le comportement adapté, un dévermineur s'ouvre et permet de corriger directement l'adaptation comme une méthode normale du système.

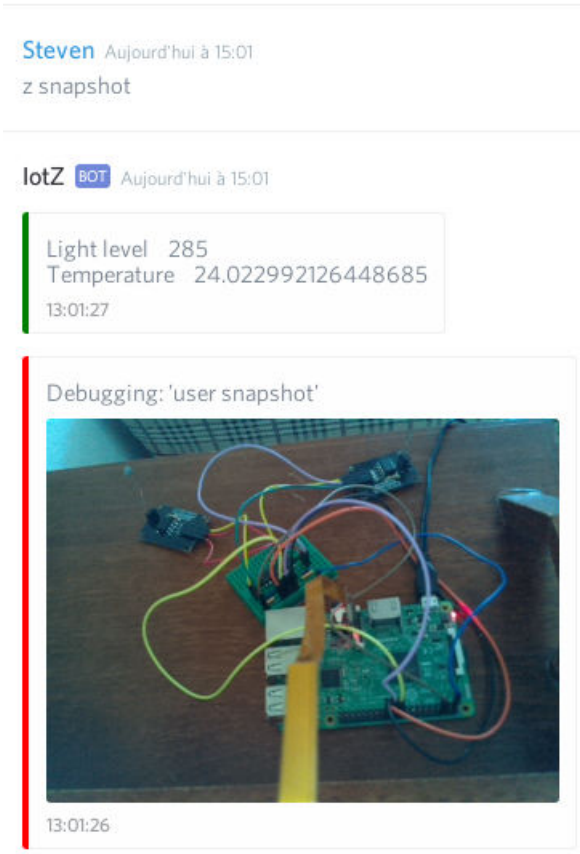


FIGURE 6.29 – Capture d'écran du serveur de discussion Discord. Le bot (*IotZ*) est désormais capable de répondre à la commande *snapshot* envoyée par un utilisateur.

Extension de l'interface du bot. Un traitement en amont filtre les messages ne contenant pas de commandes destinées au bot. Une méthode *keywords* indique à l'application les mots-clés qui permettent d'entrer dans la méthode de traitement des commandes. Une adaptation est définie pour rajouter la nouvelle commande (*snapshot*) et adapte la méthode *keywords* originelle en la remplaçant intégralement à l'exécution. La visualisation de cette adaptation est visible dans la Figure 6.27.

Utilisation et correction de la nouvelle interface. Les activateurs précédemment ajoutés au périmètre *Advanced Logging* sont supprimés et l'adaptation est modifiée pour ne plus envoyer de photographie lorsque la température monte. Le périmètre est activé par défaut, et les utilisateur-rice-s peuvent demander au bot de prendre une photographie. La méthode adaptée provoque cependant une erreur,

car le nom du serveur a été mal orthographié. Le déverminer distant de TelePharo s'ouvre sur le code de la méthode définie dans l'adaptation (Figure 6.28), et permet de corriger l'erreur dans la méthode d'adaptation. La modification du code provoque la re-compilation de la méthode directement dans l'adaptation, et le comportement de l'objet adapté est automatiquement mis à jour. Une fois le nom du serveur corrigé, le programme reprend son exécution et le bot est désormais capable de répondre à la nouvelle commande (Figure 6.29).

6.5.2 Discussion

Dans cette section nous discutons d'aspects relatifs à ce scénario. Une discussion générale fait un bilan plus global des périmètres d'adaptation à la section 6.6.

Ce scénario illustre une évolution du scénario de la section 6.4. La fonctionnalité de prise de photo devient une option du programme, accessible par l'interface du bot. L'extension de l'interface permet de rendre une fonctionnalité de déverminage disponible à la demande, alors qu'elle ne fait pas partie de la conception originale du programme.

Le périmètre d'adaptation de la section 6.4 est dynamiquement modifié pour fournir la nouvelle commande. L'adaptation est notamment complétée par de nouvelles méthodes, qui réutilisent le comportement d'adaptation défini à la section 6.4. Les activateurs sont supprimés car le périmètre ne répond plus aux mêmes contraintes. L'activation manuelle permanente du périmètre permet de rendre la nouvelle commande active par défaut. À terme, lorsque la fonctionnalité de déverminage n'est plus nécessaire, la désactivation du périmètre retire les instrumentations et restaure le comportement d'origine des objets adaptés (ici l'unique instance de l'application *IotApp*).

Le périmètre pourrait être modifié par à-coups, jusqu'à ce que la fonctionnalité de déverminage permette l'usage souhaité par les utilisateur-rice-s. Dans le cas où des exceptions sont soulevées à cause du code des adaptations, ces dernières sont traitées par le système de gestion des exceptions du programme. Dans ce scénario, le déverminer distant de TelePharo s'ouvre, permet de corriger le problème puis de relancer l'exécution à partir d'une méthode choisie dans la pile d'exécution. Ce cas est plus problématique lorsqu'il n'y a pas d'environnement de déverminage actif, une erreur risquant alors d'altérer ou d'interrompre le fonctionnement du programme.

6.6 Discussion générale et évaluation

Nous avons présenté l'infrastructure des *Debug-Scopes*, combinant l'adaptation de Lub et les groupes d'objets des Collecteurs. Nous avons présenté l'outillage des *Debug-Scopes*, et illustré son application au déverminage d'un objet connecté au travers de trois scénarios. Dans cette section, nous évaluons les *Debug-Scopes* au regard des propriétés que nous recherchons pour l'adaptation non-anticipée de comportement. Puis, nous discutons des bénéfices et des limitations issus des modèles utilisés et de leur mise en œuvre.

6.6.1 Évaluation

La table 6.1 présente l'évaluation des *Debug-Scopes* au regard des propriétés recherchées pour l'adaptation non-anticipée. Cette table reprend également l'évaluation de Lub (chapitre 4) et l'évaluation des Collecteurs (chapitre 5) au regard des mêmes propriétés. Lub satisfait la propriété d'adaptation à granularité objet mais ne fournit aucun moyen de trouver les objets à adapter. Les Collecteurs permettent de grouper des objets pendant l'exécution d'un programme, mais ne sont pas capables de les adapter. Les *Debug-Scopes* combinent les avantages de Lub et des Collecteurs pour satisfaire ces deux propriétés, permettant de collecter dynamiquement les objets à adapter. Les propriétés déjà satisfaites par Lub et par les Collecteurs (chapitres 4 et 5) sont préservées, et permettent aux *Debug-Scopes* de satisfaire toutes les propriétés désirables que nous avons définies pour l'adaptation non-anticipée de comportement dans les programmes en cours d'exécution.

● Totalement supporté ◑ Partiellement supporté ○ Non supporté

	Granularité objet	Minage d'objets	Réversibilité	Identité préservée	Flexibilité	Non-intrusivité	Paradigme objet
Lub	●	○	●	●	●	●	●
Collecteurs	○	●	●	●	●	●	●
Debug-Scopes	●	●	●	●	●	●	●

TABLE 6.1 – Positionnement des *Debug-Scopes* au regard de nos contraintes pour l'adaptation non-anticipée.

6.6.2 Bénéfices des périmètres d'adaptation

Pour déverminer un programme en cours d'exécution, la possibilité d'appliquer des modifications comportementales avec une granularité objet est alléchante. Les programmes tels que le bot *IotZ*, qui doivent fournir des services de manière ininterrompue, posent la contrainte de devoir effectuer les opérations de déverminage sans stopper leur exécution. Dans le cas d'un object connecté, ce dernier peut être déployé à un endroit difficilement atteignable, et il peut être plus intéressant de tenter une adaptation de comportement que de se déplacer pour accéder à l'appareil. Certains problèmes sont difficiles à comprendre en dehors de leur contexte d'apparition, et parfois de l'environnement direct du programme (comme dans le cas d'un objet connecté). L'accès au contexte facilite alors le processus de compréhension et de correction d'un problème.

La mise en œuvre de l'adaptation au travers des périmètres d'adaptation permet d'expérimenter de manière non-anticipée des corrections, ou de fournir de nouvelles fonctionnalités pour déverminer un programme. Les périmètres permettent de construire des visualisations personnalisées du programme en cours d'exécution, et d'accéder au contexte d'exécution d'objets particuliers. Les cas imprévus non pris en compte dans la conception du programme peuvent être explorés pour comprendre un problème ou une erreur. Une fois la source d'une erreur découverte et comprise, les périmètres permettent d'expérimenter des adaptations comportementales pour valider une hypothèse, ou pour corriger l'erreur.

L'adaptation est restreinte à un périmètre précis, qui caractérise un certain contexte d'exécution. Ce contexte est constitué d'un ensemble d'objets précis à adapter, d'une adaptation comportementale à leur appliquer ainsi que de conditions sous lesquelles activer l'adaptation. Ce périmètre permet à un-e développeur-euse de visualiser la variation comportementale, et de contrôler son application. Malgré l'outillage dédié aux périmètres, la façon de procéder et les concepts manipulés restent dans le paradigme objet (classes, objets, méthodes). L'interaction avec le code original du programme se fait avec les outils natifs de l'environnement (navigateur de classes et dévermineur de Pharo) et ne nécessitent pas d'effort de prise en main supplémentaire.

La non-intrusivité de l'adaptation permet d'explorer des variations comportementales sans risquer d'oublier des instrumentations. Le code métier des objets n'est pas modifié dans le processus, et les objets peuvent toujours être restaurés dans leur comportement d'origine.

6.6.3 Problèmes et limitations

La mise en œuvre des *Debug-Scopes* et son évaluation sont sujets à des limitations. Ces dernières mettent en lumière de nouveaux problèmes liés d'abord aux

scénarios d'évaluation présentés, ainsi qu'aux modèles et leur implémentation sur lesquels reposent les périmètres d'adaptation − nommément les adaptations gardées de Lub et les Collecteurs. Les paragraphes suivants discutent de ces limitations et des possibilités pour les dépasser.

Impact sur les performances du programme en cours d'exécution. Si les performances de Lub ont été évaluées (section 4.4.4), l'impact des collecteurs sur un programme en cours d'exécution, et de leur combinaison avec Lub dans le cadre des périmètres d'adaptation, n'a pas été étudié. Notamment, les points difficiles déjà identifiés sont la compilation des adaptations et les activations/désactivations régulières de périmètres sur des grands ensembles d'objets. L'impact sur les performances du programme est un point central qui doit être évalué dans le futur, afin d'étudier la possibilité d'une utilisation pratique des outils proposés.

Passage à l'échelle sur des problèmes plus complexes. Les cas étudiés jusqu'à présent illustrent simplement le besoin et l'application de l'adaptation non-anticipée de comportement pour le déverminage de programmes en cours d'exécution. D'une part, il manque une caractérisation des cas d'application, ceux pour lesquels les périmètres d'adaptation permettent de tirer un net avantage par rapport aux techniques classiques et ceux pour lesquels ça n'est pas le cas. Une étude sur différentes populations de développeur-euse-s avec une culture et des pratiques différentes permettrait de préciser l'apport de l'outillage en fonction de l'expérience et des domaines métiers. D'autre part, il serait intéressant d'étudier comment la solution se comporte lors d'un passage à l'échelle, sur des programmes plus complexes ou des programmes en production rencontrant des problèmes difficiles à résoudre.

Fiabilité des adaptations. Il s'agit d'un point fortement limitant : il n'y a pas de garanties qu'une adaptation ne va pas provoquer l'arrêt total du système, alors même que l'argument principal est d'instrumenter le programme sans stopper son exécution. Bien que la conception de l'adaptation soit de la responsabilité de l'utilisateur-rice, ni le modèle, ni l'outillage ne fournissent de garde-fous pour traiter les exceptions issues des erreurs utilisateur. D'autre part, il n'y a pas de cloisonnement des activations des périmètres. Un périmètre peut s'activer parce que les gardes d'adaptation sont évaluées à *true*, alors que sa configuration n'est pas terminée. C'est un problème typique rencontré en *Live Programming*, pour lequel des solutions très récentes ont été proposées [Mattis et al., 2017]. Finalement, si les exceptions dues aux adaptations peuvent être interceptées par un dévermineur, il n'y a pas non plus de garanties qu'elles soient corrigeables. Par exemple, si l'état du programme est devenu incohérent suite à l'altération de comportement, il est possible qu'il ne soit plus capable de fonctionner correctement. Ce sont des limitations dont il faut avoir

conscience lors de l'utilisation des *Debug-Scopes*, et qui doivent être pondérées suivant l'intérêt apporté − le risque peut par exemple être acceptable si l'objectif est d'investiguer un problème qui de toutes façons finira par stopper le programme.

Perte de communication avec le programme. La stabilité de la communication entre un poste de développement et un programme en cours d'exécution est un problème orthogonal à l'adaptation non-anticipée de comportement. Une perte de communication peut entraîner une fin de la session de déverminage. Il n'y a cependant pas de gestion du cas de figure dans l'outillage des *Debug-Scopes*, et l'interruption de la communication peut laisser des adaptations incomplètes ou incohérentes actives dans le programme en cours d'exécution.

Programmes multi-processus et parallélisme. Dans les scénarios étudiés, nous considérons le programme en cours d'exécution comme étant complètement mono-processus. Par exemple, même si des utilisateur-rice-s peuvent envoyer des requêtes au bot pendant que ce dernier publie des données sur le serveur de discussion, les requêtes sont mises en attente et traitées séquentiellement dans une boucle du programme. Cela simplifie l'adaptation d'objets car il n'y a pas d'accès concurrents, mais limite la solution aux applications mono-processus. Les problèmes issus du parallélisme font aujourd'hui partie des grands enjeux du déverminage [Perscheid et al., 2017]. Il s'agit donc d'un défi à relever pour l'adaptation non-anticipée d'objets pour permettre l'investigation et la correction dynamique de problèmes dans les applications multi-processus.

Capitalisation des comportements adaptés. L'application d'adaptation non-anticipée n'est pas conçue pour être définitive. Elle doit être désactivée lorsque l'exploration du programme n'est plus nécessaire ou qu'un problème a été résolu. Il n'est pas possible, avec l'outillage développé, d'intégrer les comportements d'adaptation dans une version stable du programme. Cela revêt pourtant un intérêt pratique important, car si une adaptation corrige un problème il serait possible d'automatiser une mise à jour du code de base du programme, ainsi que la génération de tests prenant en compte les nouveaux contextes possibles. Ce n'est cependant pas une simple question d'outillage. La définition des périmètres d'adaptation met en œuvre de manière implicite des concepts issus d'autres paradigmes de programmation. Les activateurs sont par exemple des réifications partielles des périmètres d'activation des *layers* de la programmation orientée contexte [Hirschfeld et al., 2008, Salvaneschi et al., 2012a], en particulier des périmètres d'activation implicites et basés sur le flot de contrôle. Il faudrait donc des modèles et des outils pour automatiser la traduction de ces concepts dans le paradigme objet du code du programme.

6.7 Conclusion

Nous avons décrit les *Debug-Scopes*, ou périmètres d'adaptation, une infrastructure pour l'adaptation non-anticipée de comportement. Cette infrastructure repose sur les adaptations gardées fournies par Lub et la collecte d'objets mise en œuvre par les Collecteurs. Nous avons présenté l'outillage associé qui permet de mettre en pratique l'adaptation dans des programmes en cours d'exécution. Cet outillage, intégré au langage Pharo, permet de spécifier une adaptation, puis de caractériser son périmètre, c'est-à-dire sous quelles conditions elle est appliquée, et à quels objets. Tandis que les collecteurs récupèrent des objets précis dans le flot de contrôle, les activateurs (réifications des gardes d'adaptation) permettent de décider si l'adaptation doit être activée ou désactivée.

Nous avons présenté une évaluation de l'outillage sur un cas d'utilisation de déverminage d'un objet connecté. Nous avons illustré l'utilisation des *Debug-Scopes* au travers de trois scénarios, dans lesquels une application mono-processus déployée sur l'objet connecté est déboguée de manière non-anticipée. L'application en cours d'exécution est un *chat-bot*, qui est adapté pour corriger un problème, rajouter des capacités de déverminage avancées non prévues et étendre son interface de communication. Ces opérations de déverminage sont réalisées à distance pendant l'exécution du programme, compte tenu des contraintes de l'application qui ne doit pas s'arrêter.

Enfin, nous avons discuté des avantages et des limitations de l'infrastructure et de son outillage, qui ouvrent la voie à des travaux futurs − et notamment en ce qui concerne l'utilisation en pratique de l'adaptation non-anticipée pour le déverminage (performances, fiabilité de l'adaptation, programmes multi-processus).

Conclusion

Nous concluons notre dissertation par un rappel des problématiques abordées dans cette thèse et de leur contexte, c'est à dire la difficulté du déverminage des programmes en cours d'exécution. Nous résumons notre proposition ainsi que nos contributions pour répondre aux problématiques de recherche de la thèse. Enfin, les limites de la proposition et de sa mise en œuvre nous permettent de dresser des perspectives de recherche.

Rappel du contexte

Dans cette thèse, nous nous sommes intéressés au déverminage de programmes à objets en cours d'exécution. Nous avons dressé un état de la pratique du déverminage et des *bugs* les plus difficiles, ainsi que des techniques de déverminage et de leur outillage, puis pointé leurs limites pour le débogage non-anticipé d'objets.

Débogage d'objets

Une difficulté majeure du débogage d'objets est de porter les capacités de déverminage au niveau d'un objet spécifique, et de pouvoir activer ces capacités au moment le plus opportun pour pouvoir observer ou reproduire un problème difficile. Par corollaire, déterminer ce moment opportun ainsi que les objets à déboguer sont des étapes clés de l'activité de déverminage. En outre, si le programme est en cours d'exécution et que son débogage n'a pas été prévu, il faut pouvoir instrumenter les objets de manière complètement non-anticipée.

L'adaptation dynamique de comportement est un outil classique pour modifier un programme de manière non-anticipée. Pour déboguer des objets avec l'adaptation de comportement, il faut donc à la fois pouvoir trouver les objets d'intérêt et pouvoir les instrumenter. Cette combinaison est difficile à trouver dans l'état de l'art de l'adaptation non-anticipée de comportement. Les modèles et solutions permettant d'atteindre ces propriétés ne le font que partiellement, et soumettent parfois le programme à des contraintes de conception fortes. Cela limite la flexibilité des instrumentations et des outils de débogage.

Problématiques de recherche

Les problématiques de recherche, telles que nous les avons formulées dans l'introduction, sont les suivantes :

1. Comment peut-on déverminer des objets de manière non-anticipée, pendant l'exécution d'un programme ?
2. Comment spécifier et obtenir les ensembles d'objets à déverminer de manière non-anticipée, pendant l'exécution du programme ?

Les travaux décrits dans cette thèse tentent de répondre à ces problématiques, et nos contributions sont résumées dans la section suivante.

Résumé des contributions

Nous avons défini *Kernel-Lub*, un patron de langage objet minimal. *Kernel-Lub* représente un modèle minimal de langage objet dynamique, qui peut être étendu par deux patrons d'extension que sont *Lub* et les *Collecteurs*. Tandis que *Lub* permet d'étendre le langage avec des capacités d'adaptation non-anticipées avec une granularité objet (problématique n^{o}1), les *Collecteurs* permettent de spécifier quels objets soumettre à l'adaptation (problématique n^{o}2). Tous les langages conformes à *Kernel-Lub* peuvent alors être étendus par ces capacités d'adaptation dynamique. *Lub* et les *Collecteurs* sont mis en œuvre au travers d'outils dédiés, ainsi que d'une infrastructure pour le déverminage non-anticipé d'objets nommée *Debug-Scopes*.

Lub. Nous avons défini *Lub*, un patron d'extension de *Kernel-Lub* pour l'adaptation non-anticipé de comportement. Les langages conformes à *Kernel-Lub* peuvent être étendus par *Lub* pour obtenir des capacités d'adaptation de comportement non-anticipée avec une granularité objet. *Lub* rajoute la notion d'*adaptation* au langage, qui permet à une instance de se comporter temporairement comme une instance d'une autre classe que la sienne.

Lub a été mis en œuvre dans le langage Pharo, au travers d'une extension de la couche réflexive de Pharo nommée *Reflectivity*. Cette extension permet à *Reflectivity* d'appliquer des instrumentations au niveau des instances des classes plutôt qu'aux classes. La mise en oeuvre de *Lub* utilise alors cette extension comme mécanisme d'adaptation.

Nous présentons une analyse des performances et de la consommation mémoire dans les programmes adaptés par la mise en œuvre Pharo de *Lub*. Cette analyse montre que, si l'impact sur les performances et la consommation mémoire n'est pas nul, un-e développeur-euse peut le considérer comme raisonnable au regard des possibilités d'adaptation de déverminage non-anticipée apportées par *Lub*. Nous avons

ensuite présenté un cas d'utilisation d'une simulation d'une flottille de drones, bloquée parce qu'un des drones n'a pas pris en compte un changement de contexte. Nous illustrons l'utilisation de *Lub* en adaptant le comportement d'instances spécifiques de la simulation. Nous corrigeons dynamiquement le problème, permettant de débloquer la simulation sans la redémarrer.

Nous validons l'aspect *patron de langage objet dynamique* en mettant en œuvre *Lub* dans le langage Python, en utilisant une autre technique réflexive nommé *Talents*.

Les Collecteurs. Nous avons défini les *Collecteurs*, un patron d'extension de *Kernel-Lub*, pour spécifier des ensembles d'objets qui seront collectés pendant l'exécution du programme. Un collecteur cible une expression du programme, et lorsque cette expression est évaluée durant l'exécution du programme, l'objet issu de cette évaluation est collecté. La collecte d'objet peut être conditionnée par du code librement défini par l'utilisateur-rice, et peut collecter des informations relatives au contexte d'exécution (par exemple le nom d'une variable ou de la méthode en cours d'exécution). Les *Collecteurs* peuvent être spécifiés et modifiés de manière non-anticipée pendant l'exécution du programme, et les groupes d'objets sont alors dynamiquement constitués et varient au cours de l'exécution. Les objets ainsi collectés peuvent être directement soumis à une adaptation de comportement, ou utilisés dans des outils de débogage.

Nous avons d'abord mis en œuvre les *Collecteurs* dans Pharo, en réutilisant l'extension de *Reflectivity* que nous avons développée pour *Lub*. Nous avons développé et intégré des outils à l'environnement de développement de Pharo, qui permettent de spécifier les *Collecteurs* à partir des outils natifs de Pharo. Il est alors possible de sélectionner une expression dans le navigateur de classe pour la soumettre à la collecte. Les objets collectés sont disponibles au travers d'une interface globale, accessible à partir de n'importe quel point du programme.

Nous avons également construit des outils de débogage spécifiques au *Collecteurs*, notamment par une extension de son modèle de mise en œuvre pour conserver un historique des objets collectés. Les objets sont ainsi copiés et conservés dans la mémoire du programme, et l'utilisateur-rice peut consulter l'historique de l'exécution d'une expression de son programme. La fonctionnalité de *replay* permet de sélectionner un objet de cet historique et de l'utiliser pour rejouer une exécution particulière d'une expression, et ainsi contrôler la valeur produite par cette expression.

Nous avons évalué la mise en œuvre des *Collecteurs* et l'outillage associé au travers de trois cas d'utilisation. Tout d'abord, nous avons utilisé les *Collecteurs* pour récupérer des objets de manière non-anticipée dans un programme en cours d'exécution, et produire des traces d'exécution. Nous avons ensuite utilisé la fonctionnalité de *replay* pour éliminer l'aspect non-déterministe du comportement d'une méthode.

Après avoir trouvé dans l'historique de collecte l'objet provoquant un comportement erratique, nous avons paramétré cet objet comme *replay* de l'expression dont il était issu. Cela nous a permis de reproduire le problème de manière déterministe, afin de l'observer et de le déboguer. Enfin, nous avons expérimenté l'utilisation de l'historique de collecte sur un *bug* dans un outil open-source de génération de documentation. Alors que les outils de débogage fournis par l'environnement de Pharo n'ont pas permis de comprendre le problème, l'historique de collecte nous a permis de trouver sa source en quelques minutes.

Pour valider l'aspect *patron de langage*, nous avons mis en œuvre les *Collecteurs* dans le langage Python. Nous avons d'abord porté *Reflectivity* dans Python, afin de pouvoir utiliser la même base de mise en œuvre. Nous avons ensuite implémenté un modèle des *Collecteurs* similaires à celui de Pharo dans Python. Seule une partie de l'outillage a été portée, pour pouvoir valider l'utilisation pratique des *Collecteurs* de manière non-anticipée dans un programme en cours d'exécution.

Debug-Scopes Nous avons décrit une infrastructure pour l'adaptation non-anticipée de comportement dans les programmes en cours d'exécution, nommée *Debug-Scopes*. Cette infrastructure repose sur la combinaison des mises en œuvre dans Pharo de *Lub* et des *Collecteurs*.

Un *scope*, ou périmètre, représente les conditions sous lesquelles doit s'appliquer une adaptation. Un périmètre comprend donc des adaptations, un ensemble d'objets collectés auxquels s'appliqueront les adaptations et des gardes d'adaptation qui définissent des contraintes d'activation des adaptations. Une garde d'adaptation est un ensemble de conditions spécifiées par l'utilisateur-rice, nommées *activateurs*. Lorsque tous les *activateurs* valident leur condition d'activation, le périmètre est activé et applique l'adaptation de comportement à tous les objets collectés.

Nous avons présenté l'outillage associé à l'infrastructure, qui permet à l'utilisateur-rice de spécifier librement le périmètre de débogage. L'adaptation est spécifiée sous forme de méthodes, qui modifieront ou ajouteront du comportement aux objets adaptés. Les *Collecteurs* sont spécifiés avec leur propre outillage, qui est intégré à celui des périmètres de débogage. Les *activateurs* sont spécifiés sous la forme de conditions exprimées par l'utilisateur-rice, sous forme de blocs de code.

Nous avons ensuite présenté une évaluation de l'outillage sur un cas d'utilisation de déverminage d'un objet connecté. Nous avons présenté trois scénarios, dans lesquels une application déployée sur l'objet connecté est déboguée à distance, pendant son exécution. Nous mettons en pratique l'utilisation de *Lub* et des *Collecteurs* au travers des périmètre de débogage pour corriger un problème, rajouter dans le programme des capacités de déverminage avancées, et étendre les fonctionnalités de l'application.

Réponse aux problématiques Dans cette thèse, nous avons étendu des langages objets dynamiques avec des capacités d'adaptation non-anticipée de comportement. Les extensions de langage, *Lub* et les *Collecteurs*, permettent respectivement d'adapter des instances spécifiques d'un programme en cours d'exécution et de spécifier librement quels sont les instances à soumettre à l'adaptation dans le flot de contrôle de ce programme. Cette combinaison ne se retrouve pas dans l'état de l'art de l'adaptation non-anticipée de comportement.

À partir de ces extensions de langage, nous avons construit des outils pour appliquer l'adaptation de comportement au débogage d'objets non-anticipé. Nous avons montré qu'avec ces outils, il était possible de mettre en pratique des techniques de débogage classiques (comme des traces d'exécution) de manière complètement non-anticipée. L'outillage permet également, grâce à l'accès au contexte réel du programme en cours d'exécution, de faciliter l'observation et la reproduction de *bugs* difficiles, par rapport au même programme arrêté et relancé dans un environnement de développement.

Perspectives

Les solutions conçues et développées dans cette thèse pour le débogage d'objets par adaptation non-anticipée de comportement sont sujettes à certaines limitations. Ces limites ont été discutées à partir de l'application de *Lub* (section 4.5.5), des *Collecteurs* (section 5.7.2) et de l'infrastructure des *Debug-Scopes* (section 6.6.3). Pour clôturer cette thèse, nous rappelons ces limites et discutons des perspectives qui sont ouvertes par nos travaux.

Performances et consommation mémoire. L'utilisation de *Lub* et des *Collecteurs* a un impact sur les performances du programme et sur sa consommation mémoire. L'impact des *Collecteurs* n'a pas été évalué, mais la mise en œuvre repose sur une base commune avec *Lub*, c'est à dire *Reflectivity*, la couche réflexive de Pharo. L'impact sur les performances et la mémoire d'un programme adapté a été évalué pour *Lub*. Premièrement, un programme adapté en cours d'exécution est plus lent qu'un programme non-adapté, et consomme également plus de mémoire. Secondement, dans la mise en œuvre développée dans cette thèse, la compilation des adaptations est effectuée directement dans le programme en cours d'exécution. Cela bloque l'exécution du programme le temps que la compilation se termine et que les objets soient migrés vers leur version adaptée.

D'une part, il est nécessaire de trouver des moyens efficaces d'exécuter du comportement adapté, et d'en limiter l'impact sur la mémoire du programme. D'autre part, des techniques de compilation *hors-ligne* pourraient être étudiées et expérimentées, pour limiter le temps d'exécution passé à compiler des adaptations. Les

adaptations seraient donc compilées sur une machine distante, et seul le code compilé serait injecté dans le programme en cours d'exécution. Cela permettrait d'expérimenter les techniques de débogage présentées dans la thèse sur des programmes avec des contraintes fortes en termes de performances et de consommation mémoire, comme les systèmes embarqués autonomes tel des drones ou des robots.

Débogage d'applications parallèles. Dans cette thèse, nous n'avons considéré que le cas des programmes mono-processus. Cependant, les *bugs* dans les programmes multi-processus font partie des plus difficiles à corriger [Perscheid et al., 2017].

Le débogage de programmes parallèles par l'adaptation non-anticipées de comportement nécessite d'étudier comment soumettre l'adaptation à des contraintes de parallélisme. Par exemple, il faut pouvoir garantir que l'adaptation du comportement d'un objet utilisé par plusieurs processus différents ne mettra pas le programme dans un état incohérent. Tout d'abord, un état de l'art de l'adaptation dans les systèmes multi-processus pourrait donner de nombreuses pistes sur les solutions existantes et les difficultés qui se posent. À partir de ce point, nous pourrions étudier comment intégrer ces solutions à notre patron de langage et à leurs outils, ou encore comment étendre ces solutions avec le patron de langage pour les enrichir avec des capacités d'adaptation non-anticipée.

Application au programmes avec des contraintes de *temps-réel*. Dans cette thèse, nous avons étudié le débogage de programmes qui doivent être corrigés sans être arrêtés. Parmi ces programmes, nous n'avons considéré que ceux qui n'ont pas de contraintes de *temps-réel*. Or, les programmes embarqués, par exemple sur des systèmes cyber-physiques, peuvent être soumis à des contraintes de *temps-réel*. Ce sont des systèmes qui entrent par définition dans la catégorie des programmes dont on ne peut pas interrompre l'exécution. Par exemple, cela pourrait être le cas pour un drone en mission qui rencontrerait un problème bloquant. Ces systèmes peuvent donc bénéficier des capacités de débogage non-anticipé décrites dans la thèse.

La variation comportementale avec *Lub* et sa contextualisation par les *activateurs* et les *Collecteurs* sont librement spécifiées par l'utilisateur-rice. Les instrumentations de code et la variation de comportement doivent donc satisfaire aux contraintes d'un système *temps-réel* pour pouvoir y être appliquées. Il n'y a donc pas de garanties que la liberté d'instrumentation soit la même selon le programme et ses contraintes. Pour pouvoir déboguer des systèmes temps-réel, il est nécessaire d'étudier comment valider l'adaptation et sa condition d'activation au regard des contraintes *temps-réel* du système auquel elle est appliquée.

Sécurité de l'adaptation. Dans les travaux développés et présentés, l'instrumentation de code et la définition de la variation comportementale sont librement spécifiées par l'utilisateur-rice. Il n'y a aucune vérification ni validation des comportements adaptés, qui peuvent alors être source de problèmes si l'utilisateur-rice a commis des erreurs. En particulier, il n'y a aucun moyen de garantir que le *lookup* d'une méthode appelée à partir du code adapté n'échouera pas. La difficulté est de pouvoir vérifier en amont la validité d'un appel de méthode ou d'un accès à un état, et ce dans le corps de toutes les méthodes appelées dans un code adapté. C'est une préoccupation connexe à l'application au débogage de programmes parallèles ou avec des contraintes de *temps-réel*, pour lesquels il faut de plus s'assurer que l'adaptation est valide au regard de ces contraintes supplémentaires (parallélisme, *temps-réel*). La validation de l'adaptation est une perspective importante pour adapter de manière fiable et stable le comportement des programmes en cours d'exécution.

Capitalisation des adaptations. Dans nos travaux, les modifications comportementales ainsi que ses instrumentation (objets à collecter, gardes d'adaptation) n'ont qu'une existence temporaire lors de l'exécution du programme adapté. Une fois que la session de déverminage est terminée, il n'y a pas de capitalisation des comportements de débogage ou des correctifs mis en place par la variation comportementale, qui sont alors perdus.

Il est important de pouvoir capitaliser ces variations comportementales soit par la sauvegardes de leur définition afin de pouvoir les réutiliser, soit par une intégration dans le code de base du programme dans le cas des correctifs. La difficulté majeure dans ce dernier cas est de traduire des concepts tels que l'activation d'adaptation ou la collecte d'objets sans polluer le code par des instructions conditionnelles. La sauvegarde des spécifications d'adaptation pourrait permettre leur réutilisation de manière automatique par des mécanismes d'auto-adaptation. Les systèmes auto-adaptatifs [Salehie and Tahvildari, 2009, Weyns, 2017] sont capables, pendant l'exécution du programme, de détecter le besoin d'adaptation, de choisir l'adaptation adéquate et de l'appliquer. Il serait intéressant d'étudier un tel mécanisme utilisant une base de *bugs* distante avec des adaptations prédéfinies, et capable d'automatiser l'application d'une adaptation lorsqu'un *bug* particulier est détecté et retrouvé dans la base. Cette base serait renseignée par les développeur-euse-s chaque fois qu'un problème étudié serait étudié et résolu par une adaptation. Les instrumentations de débogage elles-mêmes pourraient être enregistrées dans la base de données, et servir à activer automatiquement des capacités de débogage précises lorsqu'un type de problème est identifié.

Outillage et évaluation. Enfin, et pour conclure ces perspectives, l'apport pratique pour le déverminage des idées et de l'outillage développés dans cette thèse devrait être plus profondément évalué. Cela pourrait se faire selon deux axes. D'une part, par une évaluation des outils sur une base de *bugs*, et déterminer les différences dans les sessions de débogage par rapport aux techniques et outils classiques. D'autre part, par une expérience contrôlée sur des groupes de développeur-euse-s et un ensemble de *bugs* à résoudre, afin de déterminer à qui profite le plus les techniques et l'outillage décrits dans la thèse.

Bibliographie

[Agans, 2002] Agans, D. J. (2002). *Debugging : The 9 indispensable rules for finding even the most elusive software and hardware problems*. Amacom.

[Agrawal et al., 1993] Agrawal, H., DeMillo, R. A., and Spafford, E. H. (1993). Debugging with dynamic slicing and backtracking. *Software : Practice and Experience*, 23(6) :589–616.

[Agrawal and Horgan, 1990] Agrawal, H. and Horgan, J. R. (1990). Dynamic program slicing. In *ACM SIGPlan Notices*, volume 25, pages 246–256. ACM.

[Appeltauer et al., 2009] Appeltauer, M., Hirschfeld, R., Haupt, M., Lincke, J., and Perscheid, M. (2009). A comparison of context-oriented programming languages. In *International Workshop on Context-Oriented Programming*, page 6. ACM.

[Appeltauer et al., 2011] Appeltauer, M., Hirschfeld, R., Haupt, M., and Masuhara, H. (2011). Contextj : Context-oriented programming with java. *Information and Media Technologies*, 6(2) :399–419.

[Appeltauer et al., 2010] Appeltauer, M., Hirschfeld, R., Masuhara, H., Haupt, M., and Kawauchi, K. (2010). Event-specific software composition in context-oriented programming. In *International Conference on Software Composition*, pages 50–65. Springer.

[Apple, 2018] Apple (2018). XCode Debugging Tools. `https://developer.apple.com/library/content/documentation/DeveloperTools/Conceptual/debugging_with_xcode/chapters/debugging_tools.html#//apple_ref/doc/uid/TP40015022-CH8-SW11`. Accessed on : 2018-04-27.

[Arloing et al., 2016] Arloing, T., Dubois, Y., Ducasse, S., and Cassou, D. (2016). Pillar : A versatile and extensible lightweight markup language. In *Proceedings of the 11th edition of the International Workshop on Smalltalk Technologies*, page 25. ACM.

[Arya et al., 2017] Arya, K., Denniston, T., Rabkin, A., and Cooperman, G. (2017). Transition watchpoints : Teaching old debuggers new tricks. *arXiv preprint arXiv :1703.10864*.

[Bardou and Dony, 1996] Bardou, D. and Dony, C. (1996). Split objects : a disciplined use of delegation within objects. In *ACM SIGPLAN Notices*, volume 31, pages 122–137. ACM.

[Bäumer et al., 1998] Bäumer, D., Riehle, D., Siberski, W., and Wulf, M. (1998). The role object pattern. In *Washington University Dept. of Computer Science*. Citeseer.

[Becht et al., 1999] Becht, M., Gurzki, T., Klarmann, J., and Muscholl, M. (1999). Rope : Role oriented programming environment for multiagent systems. In *Cooperative Information Systems, 1999. CoopIS'99. Proceedings. 1999 IFCIS International Conference on*, pages 325–333. IEEE.

[Beller et al., 2018] Beller, M., Spruit, N., Spinellis, D., and Zaidman, A. (2018). On the dichotomy of debugging behavior among programmers. In *40th International Conference on Software Engineering, ICSE 2018, Gothenborg, Sweden*.

[Bergel et al., 2013] Bergel, A., Cassou, D., Ducasse, S., and Laval, J. (2013). *Deep Into Pharo*. Lulu. com.

[Black et al., 2010] Black, A. P., Nierstrasz, O., Ducasse, S., and Pollet, D. (2010). *Pharo by example*. Lulu. com.

[Bobrow et al., 1993] Bobrow, D. G., Gabriel, R. P., and White, J. L. (1993). Clos in context : the shape of the design space. *Object Oriented Programming : The CLOS Perspective*, pages 29–61.

[Bodden, 2011] Bodden, E. (2011). Stateful breakpoints : a practical approach to defining parameterized runtime monitors. In *Proceedings of the 19th ACM SIGSOFT symposium and the 13th European conference on Foundations of software engineering*, pages 492–495. ACM.

[Böhme et al., 2017] Böhme, M., Soremekun, E. O., Chattopadhyay, S., Ugherughe, E., and Zeller, A. (2017). Where is the bug and how is it fixed ? an experiment with practitioners. In *Proceedings of the 2017 11th Joint Meeting on Foundations of Software Engineering*, pages 117–128. ACM.

[Bonér, 2004] Bonér, J. (2004). Aspectwerkz-dynamic aop for java. In *Invited talk at 3rd International Conference on Aspect-Oriented Software Development (AOSD)*. Citeseer.

[Boothe, 2000] Boothe, B. (2000). Efficient algorithms for bidirectional debugging. *ACM SIGPLAN Notices*, 35(5) :299–310.

[Bouraqadi, 2000] Bouraqadi, N. (2000). Concern oriented programming using reflection. In *Workshop on Advanced Separation of Concerns–OOSPLA*, volume 2000.

[Bouraqadi Saadani, 1999] Bouraqadi Saadani, M. N. (1999). *Un MOP Smalltalk pour l'étude de la composition et de la compatibilité des métaclasses. Application à la programmation par aspects*. PhD thesis.

[Bouraqadi-Saâdani et al., 1998] Bouraqadi-Saâdani, N., Ledoux, T., and Rivard, F. (1998). Safe metaclass programming. In *ACM SIGPLAN Notices*, volume 33, pages 84–96. ACM.

[Brant et al., 1998] Brant, J., Foote, B., Johnson, R. E., and Roberts, D. (1998). Wrappers to the rescue. In *European Conference on Object-Oriented Programming*, pages 396–417. Springer.

[Carré, 1989] Carré, B. (1989). *Méthodologie orientée objet pour la représentation des connaissances : concepts de point de vue, de représentation multiple et évolutive d'objet*. PhD thesis, Lille 1.

[Chern and De Volder, 2007] Chern, R. and De Volder, K. (2007). Debugging with control-flow breakpoints. In *Proceedings of the 6th international conference on Aspect-oriented software development*, pages 96–106. ACM.

[Chiş et al., 2015] Chiş, A., Denker, M., Gîrba, T., and Nierstrasz, O. (2015). Practical domain-specific debuggers using the moldable debugger framework. *Computer Languages, Systems & Structures*, 44 :89–113.

[Cleve and Zeller, 2005] Cleve, H. and Zeller, A. (2005). Locating causes of program failures. In *Proceedings of the 27th international conference on Software engineering*, pages 342–351. ACM.

[Costa-Soria et al., 2011] Costa-Soria, C., Pérez, J., and Carsí, J. Á. (2011). An aspect-oriented approach for supporting autonomic reconfiguration of software architectures. *Informatica*, 35(1).

[Costanza and Hirschfeld, 2005] Costanza, P. and Hirschfeld, R. (2005). Language constructs for context-oriented programming : an overview of contextl. In *Proceedings of the 2005 symposium on Dynamic languages*, pages 1–10. ACM.

[Costiou et al., 2016] Costiou, S., Kerboeuf, M., Cavarlé, G., and Plantec, A. (2016). Lub : a dsl for dynamic context oriented programming. In *Proceedings of the 11th edition of the International Workshop on Smalltalk Technologies*, page 13. ACM.

[Costiou et al., 2018a] Costiou, S., Kerboeuf, M., Cavarlé, G., and Plantec, A. (2018a). Lub : A pattern for fine grained behavior adaptation at runtime. *Science of Computer Programming*, 161 :149–171.

[Costiou et al., 2018b] Costiou, S., Kerboeuf, M., Plantec, A., and Denker, M. (2018b). Collectors. In *Conference Companion of the 2nd International Conference on Art, Science, and Engineering of Programming*, pages 144–152. ACM.

[Costiou et al., 2018c] Costiou, S., Toullec, C., Kerboeuf, M., and Plantec, A. (2018c). Back-in-time inspectors : an implementation with collectors (to appear). In *Proceedings of the 13th edition of the International Workshop on Smalltalk Technologies (to appear)*, page 13. ACM.

[Cotroneo et al., 2014] Cotroneo, D., Natella, R., Pietrantuono, R., and Russo, S. (2014). A survey of software aging and rejuvenation studies. *ACM Journal on Emerging Technologies in Computing Systems (JETC)*, 10(1) :8.

[Cotroneo et al., 2006] Cotroneo, D., Orlando, S., and Russo, S. (2006). Failure classification and analysis of the java virtual machine. In *Distributed Computing Systems, 2006. ICDCS 2006. 26th IEEE International Conference on*, pages 17–17. IEEE.

[de Pina, 2016] de Pina, L. G. G. (2016). *Practical Dynamic Software Updating*. PhD thesis, INSTITUTO SUPERIOR TECNICO.

[Denker, 2008] Denker, M. (2008). *Sub-method Structural and Behavioral Reflection*. Lulu. com.

[Denker et al., 2007] Denker, M., Ducasse, S., Lienhard, A., and Marschall, P. (2007). Sub-method reflection. In *TOOLS Europe 2007*, volume 6, pages 231–251. JOT.

[Dony et al., 1998] Dony, C., Malenfant, J., and Bardou, D. (1998). Les langages a prototypes. *Langages et Mod eles d'Objets. INRIA-Collection Didactique*.

[Ducasse, 1999] Ducasse, S. (1999). Evaluating message passing control techniques in smalltalk. *Journal of Object Oriented Programming*, 12 :39–50.

[Ducasse et al., 2006] Ducasse, S., Nierstrasz, O., Schärli, N., Wuyts, R., and Black, A. P. (2006). Traits : A mechanism for fine-grained reuse. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 28(2) :331–388.

[Duncan, 2003] Duncan, S. (2003). Component software : Beyond object-oriented programming. *Software Quality Professional*, 5(4) :42.

[Dupriez et al., 2017] Dupriez, T., Polito, G., and Ducasse, S. (2017). Analysis and exploration for new generation debuggers. In *Proceedings of the 12th edition of the International Workshop on Smalltalk Technologies*, page 5. ACM.

[Eclipse, 2018a] Eclipse (2018a). Eclipse AspectJ. `https://www.eclipse.org/aspectj/`. Accessed on : 2018-07-26.

[Eclipse, 2018b] Eclipse (2018b). Eclipse Stacktrace Console. `https://help.eclipse.org/mars/index.jsp?topic=%2Forg.eclipse.jdt.doc.user%2Freference%2Fviews%2Fconsole%2Fref-stacktrace_console.htm`. Accessed on : 2018-07-23.

[Eclipse, 2018c] Eclipse (2018c). Java Eclipse Debugger. `https://help.eclipse.org/neon/index.jsp?topic=%2Forg.eclipse.jdt.doc.user%2Ftasks%2Ftask-running_and_debugging.htm&cp=1_3_6`. Accessed on : 2018-04-26.

[Eisenstadt, 1997] Eisenstadt, M. (1997). My hairiest bug war stories. *Communications of the ACM*, 40(4) :30–37.

[Eugster, 2006] Eugster, P. (2006). Uniform proxies for java. *ACM SIGPLAN Notices*, 41(10) :139–152.

[Fabresse et al., 2006] Fabresse, L., Dony, C., and Huchard, M. (2006). Unanticipated connection of components based on their state changes notifications. In *EECC : Evaluation and Evolution of Component Composition*.

[Fabry and Galdames, 2014] Fabry, J. and Galdames, D. (2014). Phantom : a modern aspect language for pharo smalltalk. *Software : Practice and Experience*, 44(4) :393–412.

[Feldman and Brown, 1988] Feldman, S. I. and Brown, C. B. (1988). Igor : A system for program debugging via reversible execution. In *ACM Sigplan Notices*, volume 24, pages 112–123. ACM.

[Felgentreff et al., 2015] Felgentreff, T., Lincke, J., Hirschfeld, R., and Thamsen, L. (2015). Lively groups : shared behavior in a world of objects without classes or prototypes. In *Proceedings of the Workshop on Future Programming*, pages 15–22. ACM.

[Filman et al., 2004] Filman, R., Haupt, M., Mehner, K., and Mezini, M. (2004). Daw : Dynamic aspects workshop. In *DAW : Dynamic Aspects Workshop*.

[Foote and Johnson, 1989] Foote, B. and Johnson, R. E. (1989). Reflective facilities in smalltalk-80. In *ACM Sigplan Notices*, volume 24, pages 327–335. ACM.

[Fowler, 1997] Fowler, M. (1997). Dealing with roles. In *Proceedings of PLoP*, volume 97.

[Fowler, 2004] Fowler, M. (2004). *UML distilled : a brief guide to the standard object modeling language*. Addison-Wesley Professional.

[Gamma et al., 1993] Gamma, E., Helm, R., Johnson, R., and Vlissides, J. (1993). Design patterns : Abstraction and reuse of object-oriented design. In *European Conference on Object-Oriented Programming*, pages 406–431. Springer.

[Gjerlufsen et al., 2009] Gjerlufsen, T., Ingstrup, M., and Olsen, J. W. (2009). Mirrors of meaning : Supporting inspectable runtime models. *Computer*, 42(10).

[Goldberg and Robson, 1983] Goldberg, A. and Robson, D. (1983). *Smalltalk-80 : the language and its implementation*. Addison-Wesley Longman Publishing Co., Inc.

[Goldberg, 1984] Goldberg, A. J. (1984). Smalltalk-80 : the interactive programming environment.

[González et al., 2013] González, S., Mens, K., Colacioiu, M., and Cazzola, W. (2013). Context traits : dynamic behaviour adaptation through run-time trait recomposition. In *Proceedings of the 12th annual international conference on Aspect-oriented software development*, pages 209–220. ACM.

[Gottbrath, 2008] Gottbrath, C. (2008). Reverse debugging with the totalview debugger. In *Cray User Group Conference*, pages 5–8. Citeseer.

[Gowing and Cahill, 1996] Gowing, B. and Cahill, V. (1996). *Meta-object protocols for C++ : The Iguana approach*. Citeseer.

[Graversen and Osterbye, 2003] Graversen, K. B. and Osterbye, K. (2003). Implementation of a role language for object-specific dynamic separation of concerns. In *AOSD03 Workshop on Software-engineering Properties of Languages for Aspect Technologies*. Citeseer.

[Gray, 1986] Gray, J. (1986). Why do computers stop and what can be done about it ? In *Symposium on reliability in distributed software and database systems*, pages 3–12. Los Angeles, CA, USA.

[Grottke et al., 2008] Grottke, M., Matias, R., and Trivedi, K. S. (2008). The fundamentals of software aging. In *Software Reliability Engineering Workshops, 2008. ISSRE Wksp 2008. IEEE International Conference on*, pages 1–6. Ieee.

[Grottke and Trivedi, 2007] Grottke, M. and Trivedi, K. S. (2007). Fighting bugs : Remove, retry, replicate, and rejuvenate. *Computer*, 40(2).

[Gu et al., 2014] Gu, T., Cao, C., Xu, C., Ma, X., Zhang, L., and Lü, J. (2014). Low-disruptive dynamic updating of java applications. *Information and Software Technology*, 56(9) :1086–1098.

[Gulzar et al., 2016a] Gulzar, M. A., Interlandi, M., Condie, T., and Kim, M. (2016a). Bigdebug : Interactive debugger for big data analytics in apache spark. In *Proceedings of the 2016 24th ACM SIGSOFT International Symposium on Foundations of Software Engineering*, pages 1033–1037. ACM.

[Gulzar et al., 2017] Gulzar, M. A., Interlandi, M., Condie, T., and Kim, M. (2017). Debugging big data analytics in spark with bigdebug. In *Proceedings of the 2017 ACM International Conference on Management of Data*, pages 1627–1630. ACM.

[Gulzar et al., 2016b] Gulzar, M. A., Interlandi, M., Yoo, S., Tetali, S. D., Condie, T., Millstein, T., and Kim, M. (2016b). Bigdebug : Debugging primitives for interactive big data processing in spark. In *Proceedings of the 38th International Conference on Software Engineering*, pages 784–795. ACM.

[Hammer, 2009] Hammer, M. (2009). How to touch a running system-reconfiguration of stateful components.

[Hammer and Knapp, 2010] Hammer, M. and Knapp, A. (2010). Correct execution of reconfiguration for stateful components. *Electronic Notes in Theoretical Computer Science*, 260 :91–108.

[He et al., 2006] He, C., Nie, Z., Li, B., Cao, L., and He, K. (2006). Rava : Designing a java extension with dynamic object roles. In *Engineering of Computer Based Systems, 2006. ECBS 2006. 13th Annual IEEE International Symposium and Workshop on*, pages 7–pp. IEEE.

[Herrmann, 2005] Herrmann, S. (2005). Programming with roles in objectteams/-java. In *proc. AAAI Fall Symposium*.

[Hinkle et al., 1993] Hinkle, B., Jones, V., and Johnson, R. E. (1993). Debugging objects. In *The Smalltalk Report*. Citeseer.

[Hirschfeld, 2002] Hirschfeld, R. (2002). Aspects-aspect-oriented programming with squeak. In *Net. ObjectDays : International Conference on Object-Oriented and Internet-Based Technologies, Concepts, and Applications for a Networked World*, pages 216–232. Springer.

[Hirschfeld et al., 2007] Hirschfeld, R., Costanza, P., and Haupt, M. (2007). An introduction to context-oriented programming with contexts. In *International Summer School on Generative and Transformational Techniques in Software Engineering*, pages 396–407. Springer.

[Hirschfeld et al., 2008] Hirschfeld, R., Costanza, P., and Nierstrasz, O. M. (2008). Context-oriented programming. *Journal of Object technology*, 7(3) :125–151.

[Hofer et al., 2006] Hofer, C., Denker, M., and Ducasse, S. (2006). Design and implementation of a backward-in-time debugger. In *NODe 2006*, pages 17–32. GI.

[Infante and Bergel, 2017] Infante, A. and Bergel, A. (2017). Object equivalence : Revisiting object equality profiling (an experience report). *SIGPLAN Not.*, 52(11) :27–38.

[Ingalls et al., 2016] Ingalls, D., Felgentreff, T., Hirschfeld, R., Krahn, R., Lincke, J., Röder, M., Taivalsaari, A., and Mikkonen, T. (2016). A world of active objects for work and play : the first ten years of lively. In *Proceedings of the 2016 ACM International Symposium on New Ideas, New Paradigms, and Reflections on Programming and Software*, pages 238–249. ACM.

[Ingalls et al., 2008] Ingalls, D., Palacz, K., Uhler, S., Taivalsaari, A., and Mikkonen, T. (2008). The lively kernel a self-supporting system on a web page. In *Self-Sustaining Systems*, pages 31–50. Springer.

[Jones, 2003] Jones, J. (2003). Abstract syntax tree implementation idioms. In *Proceedings of the 10th conference on pattern languages of programs (plop2003)*, pages 1–10.

[Kamina et al., 2011] Kamina, T., Aotani, T., and Masuhara, H. (2011). Eventcj : a context-oriented programming language with declarative event-based context transition. In *Proceedings of the tenth international conference on Aspect-oriented software development*, pages 253–264. ACM.

[Kamina et al., 2016] Kamina, T., Aotani, T., and Masuhara, H. (2016). Generalized layer activation mechanism for context-oriented programming. In *Transactions on Modularity and Composition I*, pages 123–166. Springer.

[Kamina and Tamai, 2008] Kamina, T. and Tamai, T. (2008). Flexible object adaptation for java-like languages. In *Proceedings of the 10th Workshop on Formal Techniques for Java-like Programs (FTfJP 2008)*, pages 63–76.

[Kamina and Tamai, 2009] Kamina, T. and Tamai, T. (2009). Towards safe and flexible object adaptation. In *International Workshop on Context-Oriented Programming*, page 4. ACM.

[Katz and Anderson, 1987] Katz, I. R. and Anderson, J. R. (1987). Debugging : An analysis of bug-location strategies. *Human-Computer Interaction*, 3(4) :351–399.

[Keeney, 2004] Keeney, J. (2004). *Completely unanticipated dynamic adaptation of software*. PhD thesis, Trinity College Dublin.

[Keeney and Cahill, 2003] Keeney, J. and Cahill, V. (2003). Chisel : A policy-driven, context-aware, dynamic adaptation framework. In *Policies for Distributed Systems and Networks, 2003. Proceedings. POLICY 2003. IEEE 4th International Workshop on*, pages 3–14. IEEE.

[Kendall and Heath, 1998] Kendall, E. A. and Heath, M. (1998). Agent roles and role models : New abstractions for intelligent agent system analysis and design. In *International Workshop on Intelligent Agents in Information and Process Management*.

[Kerrisk, 2010] Kerrisk, M. (2010). *The Linux programming interface : a Linux and UNIX system programming handbook*. No Starch Press.

[Kiczales et al., 1991] Kiczales, G., Des Rivieres, J., and Bobrow, D. G. (1991). *The art of the metaobject protocol*. MIT press.

[Kiczales et al., 2001a] Kiczales, G., Hilsdale, E., Hugunin, J., Kersten, M., Palm, J., and Griswold, W. (2001a). Getting started with aspectj. *Communications of the ACM*, 44(10) :59–65.

[Kiczales et al., 2001b] Kiczales, G., Hilsdale, E., Hugunin, J., Kersten, M., Palm, J., and Griswold, W. G. (2001b). An overview of aspectj. In *European Conference on Object-Oriented Programming*, pages 327–354. Springer.

[Kiczales et al., 1997] Kiczales, G., Lamping, J., Mendhekar, A., Maeda, C., Lopes, C., Loingtier, J.-M., and Irwin, J. (1997). Aspect-oriented programming. In *European conference on object-oriented programming*, pages 220–242. Springer.

[Kniesel, 1999] Kniesel, G. (1999). Type-safe delegation for run-time component adaptation. In *European Conference on Object-Oriented Programming*, pages 351–366. Springer.

[Kniesel et al., 2002] Kniesel, G., Noppen, J., Mens, T., and Buckley, J. (2002). Unanticipated software evolution. In *European Conference on Object-Oriented Programming*, pages 92–106. Springer.

[Knuth, 1989] Knuth, D. E. (1989). The errors of tex. *Software : Practice and Experience*, 19(7) :607–685.

[Ko and Myers, 2008] Ko, A. and Myers, B. (2008). Debugging reinvented. In *Software Engineering, 2008. ICSE'08. ACM/IEEE 30th International Conference on*, pages 301–310. IEEE.

[Ko and Myers, 2004] Ko, A. J. and Myers, B. A. (2004). Designing the whyline : a debugging interface for asking questions about program behavior. In *Proceedings of the SIGCHI conference on Human factors in computing systems*, pages 151–158. ACM.

[Korel and Laski, 1988] Korel, B. and Laski, J. (1988). Dynamic program slicing. *Information processing letters*, 29(3) :155–163.

[Krahn et al., 2012] Krahn, R., Lincke, J., and Hirschfeld, R. (2012). Efficient layer activation in context js. In *2012 10th International Conference on Creating, Connecting and Collaborating through Computing*, pages 76–83. IEEE.

[Kudriashov, 2016] Kudriashov, D. (2016). Ghost object mutation. `https://dionisiydk.blogspot.com/2016/04/playing-with-objectvirus.html`. Accessed on : 2018-08-16.

[Kühn et al., 2014] Kühn, T., Leuthäuser, M., Götz, S., Seidl, C., and Aßmann, U. (2014). A metamodel family for role-based modeling and programming languages. In *International Conference on Software Language Engineering*, pages 141–160. Springer.

[Lau and Wang, 2005] Lau, K.-K. and Wang, Z. (2005). A taxonomy of software component models. In *null*, pages 88–95. IEEE.

[Lau and Wang, 2007] Lau, K.-K. and Wang, Z. (2007). Software component models. *IEEE Transactions on software engineering*, 33(10).

[Layman et al., 2013] Layman, L., Diep, M., Nagappan, M., Singer, J., Deline, R., and Venolia, G. (2013). Debugging revisited : Toward understanding the debugging needs of contemporary software developers. In *Empirical Software Engineering and Measurement, 2013 ACM/IEEE International Symposium on*, pages 383–392. IEEE.

[Lehmann et al., 2016] Lehmann, S., Felgentreff, T., Lincke, J., Rein, P., and Hirschfeld, R. (2016). Reactive object queries. In *Constrained and Reactive Objects Workshop (CROW)*.

[Lencevicius, 2000] Lencevicius, R. (2000). On-the-fly query-based debugging with examples. *arXiv preprint cs/0011021*.

[Lencevicius et al., 1997] Lencevicius, R., Hölzle, U., and Singh, A. K. (1997). Query-based debugging of object-oriented programs. In *ACM SIGPLAN Notices*, volume 32, pages 304–317. ACM.

[Lencevicius et al., 1999] Lencevicius, R., Hölzle, U., and Singh, A. K. (1999). Dynamic query-based debugging. In *European Conference on Object-Oriented Programming*, pages 135–160. Springer.

[Lencevicius et al., 2003] Lencevicius, R., Hölzle, U., and Singh, A. K. (2003). Dynamic query-based debugging of object-oriented programs. *Automated Software Engineering*, 10(1) :39–74.

[Lewis, 2003] Lewis, B. (2003). Debugging backwards in time. *arXiv preprint cs/0310016*.

[Lieberman, 1986] Lieberman, H. (1986). Using prototypical objects to implement shared behavior in object-oriented systems. *ACM Sigplan Notices*, 21(11) :214–223.

[Lienhard et al., 2006] Lienhard, A., Ducasse, S., Gırba, T., and Nierstrasz, O. (2006). Capturing how objects flow at runtime. In *Proceedings International Workshop on Program Comprehension through Dynamic Analysis (PCODA 2006)*, pages 39–43.

[Lienhard et al., 2009] Lienhard, A., Fierz, J., and Nierstrasz, O. (2009). Flow-centric, back-in-time debugging. In *International Conference on Objects, Components, Models and Patterns*, pages 272–288. Springer.

[Lienhard et al., 2008] Lienhard, A., Gîrba, T., and Nierstrasz, O. (2008). Practical object-oriented back-in-time debugging. In *European Conference on Object-Oriented Programming*, pages 592–615. Springer.

[Lincke et al., 2011] Lincke, J., Appeltauer, M., Steinert, B., and Hirschfeld, R. (2011). An open implementation for context-oriented layer composition in contextjs. *Science of Computer Programming*, 76(12) :1194–1209.

[Maier, 1998] Maier, M. W. (1998). Architecting principles for systems-of-systems. *Systems Engineering : The Journal of the International Council on Systems Engineering*, 1(4) :267–284.

[Malenfant et al., 1996] Malenfant, J., Jacques, M., and Demers, F. N. (1996). A tutorial on behavioral reflection and its implementation. In *Proceedings of the Reflection*, volume 96, pages 1–20.

[Maloney and Smith, 1995] Maloney, J. H. and Smith, R. B. (1995). Directness and liveness in the morphic user interface construction environment. In *Proceedings of the 8th annual ACM symposium on User interface and software technology*, pages 21–28. ACM.

[Marra, 2017] Marra, M. (2017). Idra : An out-of-place debugger for non-stoppable applications.

[Marra, 2018] Marra, M. (2018). Debugging support for big data processing applications. In *Conference Companion of the 2nd International Conference on Art, Science, and Engineering of Programming*, pages 241–242. ACM.

[Marra et al., 2017] Marra, M., Boix, E. G., Costiou, S., Kerboeuf, M., Plantec, A., Polito, G., and Ducasse, S. (2017). Debugging cyber-physical systems with pharo : An experience report. In *Proceedings of the 12th edition of the International Workshop on Smalltalk Technologies*, page 8. ACM.

[Marra et al., 2018] Marra, M., Polito, G., and Boix, E. G. (2018). Out-of-place Debugging : A debugging architecture to reduce debugging interference. *To appear in The Art, Science, and Engineering of Programming*, 3(2).

[Martinez et al., 2015] Martinez, S., Dagnat, F., and Buisson, J. (2015). Pymoult : On-line updates for python programs. In *ICSEA 2015 : 10th International Conference on Software Engineering Advances*, pages 80–85.

[Martinez et al., 2013] Martinez, S., Dagnat, F., Buisson, J., et al. (2013). Prototyping dsu techniques using python. In *HotSWUp*.

[Mattis et al., 2017] Mattis, T., Rein, P., and Hirschfeld, R. (2017). Edit transactions : Dynamically scoped change sets for controlled updates in live programming. *arXiv preprint arXiv :1703.10862*.

[McDirmid, 2007] McDirmid, S. (2007). Living it up with a live programming language. In *ACM SIGPLAN Notices*, volume 42, pages 623–638. ACM.

[McDirmid, 2013] McDirmid, S. (2013). Usable live programming. In *Proceedings of the 2013 ACM international symposium on New ideas, new paradigms, and reflections on programming & software*, pages 53–62. ACM.

[Microsoft, 2018a] Microsoft (2018a). Visual Studio Advanced BreakPoints. `https://docs.microsoft.com/en-us/visualstudio/debugger/using-breakpoints#BKMK_Specify_a_breakpoint_condition_using_a_code_expression`. Accessed on : 2018-04-26.

[Microsoft, 2018b] Microsoft (2018b). Visual studio remote debugging. `https://msdn.microsoft.com/en-us/library/y7f5zaaa.aspx`. Accessed on : 2018-07-22.

[Miedes and Munoz-Escoı, 2012] Miedes, E. and Munoz-Escoı, F. D. (2012). Dynamic software update. *Instituto Universitario Mixto Tecnológico de Informática, Universitat Politècnica de València, Technical Report ITI-SIDI-2012/004*.

[Milojkovic et al., 2017] Milojkovic, N., Ghafari, M., and Nierstrasz, O. (2017). It's duck (typing) season ! In *Program Comprehension (ICPC), 2017 IEEE/ACM 25th International Conference on*, pages 312–315. IEEE.

[Misherghi and Su, 2006] Misherghi, G. and Su, Z. (2006). Hdd : hierarchical delta debugging. In *Proceedings of the 28th international conference on Software engineering*, pages 142–151. ACM.

[Monpratarnchai and Tetsuo, 2008] Monpratarnchai, S. and Tetsuo, T. (2008). The design and implementation of a role model based language, epsilonj. In *Electrical Engineering/Electronics, Computer, Telecommunications and Information Technology, 2008. ECTI-CON 2008. 5th International Conference on*, volume 1, pages 37–40. IEEE.

[Murphy et al., 2006] Murphy, G. C., Kersten, M., and Findlater, L. (2006). How are java software developers using the elipse ide ? *IEEE software*, 23(4) :76–83.

[Neamtiu et al., 2006] Neamtiu, I., Hicks, M., Stoyle, G., and Oriol, M. (2006). *Practical dynamic software updating for C*, volume 41. ACM.

[Nierstrasz et al., 2009a] Nierstrasz, O., Denker, M., and Renggli, L. (2009a). Model-centric, context-aware software adaptation. In *Software Engineering for Self-Adaptive Systems*, pages 128–145. Springer.

[Nierstrasz et al., 2009b] Nierstrasz, O., Ducasse, S., and Pollet, D. (2009b). *Squeak by example*. Lulu. com.

[Oriol, 2004] Oriol, M. (2004). *An approach to the dynamic evolution of software systems*. PhD thesis, University of Geneva.

[Papoulias et al., 2015] Papoulias, N., Bouraqadi, N., Fabresse, L., Ducasse, S., and Denker, M. (2015). Mercury : Properties and design of a remote debugging solution using reflection. *The Journal of Object Technology*, 14(2) :36.

[Papoulias et al., 2017] Papoulias, N., Denker, M., Ducasse, S., and Fabresse, L. (2017). End-user abstractions for meta-control : Reifying the reflectogram. *Science of Computer Programming*, 140 :2–16.

[Parnas, 1994] Parnas, D. L. (1994). Software aging. In *Proceedings of the 16th international conference on Software engineering*, pages 279–287. IEEE Computer Society Press.

[PDB, 2018] PDB (2018). Python Debugger. `https://docs.python.org/3.8/library/pdb.html`. Accessed on : 2018-07-22.

[Peck et al., 2015] Peck, M. M., Bouraqadi, N., Fabresse, L., Denker, M., and Teruel, C. (2015). Ghost : A uniform and general-purpose proxy implementation. *Science of Computer Programming*, 98(3) :339–359.

[Perscheid et al., 2017] Perscheid, M., Siegmund, B., Taeumel, M., and Hirschfeld, R. (2017). Studying the advancement in debugging practice of professional software developers. *Software Quality Journal*, 25(1) :83–110.

[Piechnick et al., 2012] Piechnick, C., Richly, S., Götz, S., Wilke, C., and Aßmann, U. (2012). Using role-based composition to support unanticipated, dynamic adaptation-smart application grids. *Proceedings of ADAPTIVE*, pages 93–102.

[Pina and Hicks, 2013] Pina, L. and Hicks, M. (2013). Rubah : Efficient, general-purpose dynamic software updating for java. In *HotSWUp*.

[Pluquet et al., 2009] Pluquet, F., Langerman, S., and Wuyts, R. (2009). Executing code in the past : efficient in-memory object graph versioning. In *ACM SIGPLAN Notices*, volume 44, pages 391–408. ACM.

[Popovici et al., 2002] Popovici, A., Gross, T., and Alonso, G. (2002). Dynamic weaving for aspect-oriented programming. In *Proceedings of the 1st international conference on Aspect-oriented software development*, pages 141–147. ACM.

[Pothier and Tanter, 2009] Pothier, G. and Tanter, É. (2009). Back to the future : Omniscient debugging. *IEEE software*, 26(6).

[Pothier and Tanter, 2011] Pothier, G. and Tanter, É. (2011). Summarized trace indexing and querying for scalable back-in-time debugging. In *European Conference on Object-Oriented Programming*, pages 558–582. Springer.

[Pothier et al., 2007] Pothier, G., Tanter, É., and Piquer, J. (2007). Scalable omniscient debugging. *ACM SIGPLAN Notices*, 42(10) :535–552.

[Python, 2017] Python (2017). Python 3.4.5 documentation. `https://docs.python.org/3.4/`. Accessed : 2017-02-06.

[Rajan and Sullivan, 2003] Rajan, H. and Sullivan, K. (2003). Eos : instance-level aspects for integrated system design. *ACM SIGSOFT Software Engineering Notes*, 28(5) :297–306.

[Ramson et al., 2017] Ramson, S., Lincke, J., and Hirschfeld, R. (2017). The declarative nature of implicit layer activation. In *Proceedings of the 9th International Workshop on Context-Oriented Programming*, pages 7–16. ACM.

[Raymond and Steele, 1996] Raymond, E. S. and Steele, G. L. (1996). *The new hacker's dictionary*. Mit Press.

[Redmond and Cahill, 2000] Redmond, B. and Cahill, V. (2000). Iguana/j : Towards a dynamic and efficient reflective architecture for java. In *ECOOP 2000 Workshop on Reflection and Metalevel Architectures*.

[Redmond and Cahill, 2002] Redmond, B. and Cahill, V. (2002). Supporting unanticipated dynamic adaptation of application behaviour. In *European Conference on Object-Oriented Programming*, pages 205–230. Springer.

[Rein et al., 2017] Rein, P., Ramson, S., Lincke, J., Felgentreff, T., and Hirschfeld, R. (2017). Group-based behavior adaptation mechanisms in object-oriented systems. *IEEE Software*, 34(6) :78–82.

[Rein et al., 2018] Rein, P., Ramson, S., Lincke, J., Hirschfeld, R., and Pape, T. (2018). Exploratory and Live, Programming and Coding. *The Art, Science, and Engineering of Programming*, 3(1) :1 :1–1 :33.

[Ressia et al., 2012] Ressia, J., Bergel, A., and Nierstrasz, O. (2012). Object-centric debugging. In *Proceedings of the 34th International Conference on Software Engineering*, pages 485–495. IEEE Press.

[Ressia et al., 2014] Ressia, J., Gîrba, T., Nierstrasz, O., Perin, F., and Renggli, L. (2014). Talents : an environment for dynamically composing units of reuse. *Software : Practice and Experience*, 44(4) :413–432.

[Ressia et al., 2010] Ressia, J., Renggli, L., Gîrba, T., and Nierstrasz, O. (2010). Run-time evolution through explicit meta-objects. In *Proceedings of the 5th Workshop on Models@ run. time at the ACM/IEEE 13th International Conference on Model Driven Engineering Languages and Systems (MODELS 2010)*, pages 37–48.

[Ronsse and De Bosschere, 1999] Ronsse, M. and De Bosschere, K. (1999). Recplay : a fully integrated practical record/replay system. *ACM Transactions on Computer Systems (TOCS)*, 17(2) :133–152.

[Rosà et al., 2016] Rosà, A., Zheng, Y., Sun, H., Javed, O., and Binder, W. (2016). Adaptable runtime monitoring for the java virtual machine. In *International Symposium on Leveraging Applications of Formal Methods*, pages 531–546. Springer.

[Rosenberg, 1996] Rosenberg, J. B. (1996). *How debuggers work : algorithms, data structures, and architecture*, volume 8. John Wiley.

[Röthlisberger et al., 2008] Röthlisberger, D., Denker, M., and Tanter, É. (2008). Unanticipated partial behavioral reflection : Adapting applications at runtime. *Computer Languages, Systems & Structures*, 34(2-3) :46–65.

[Rudametkin Ivey, 2013] Rudametkin Ivey, W. A. (2013). *Robusta : une approche pour la construction d'applications dynamiques*. PhD thesis, Grenoble.

[Salehie and Tahvildari, 2009] Salehie, M. and Tahvildari, L. (2009). Self-adaptive software : Landscape and research challenges. *ACM transactions on autonomous and adaptive systems (TAAS)*, 4(2) :14.

[Salvaneschi et al., 2012a] Salvaneschi, G., Ghezzi, C., and Pradella, M. (2012a). Context-oriented programming : A software engineering perspective. *Journal of Systems and Software*, 85(8) :1801–1817.

[Salvaneschi et al., 2012b] Salvaneschi, G., Ghezzi, C., and Pradella, M. (2012b). Contexterlang : introducing context-oriented programming in the actor model. In *Proceedings of the 11th annual international conference on Aspect-oriented Software Development*, pages 191–202. ACM.

[Sanen et al., 2005] Sanen, F., Steegmans, E., Picard, N., Joosen, W., and Holvoet, T. (2005). Using roles and aspects for designing and implementing dynamic adaptations.

[Schneider et al., 2015a] Schneider, J.-P., Champeau, J., Lagadec, L., and Senn, E. (2015a). Role framework to support collaborative virtual prototyping of system of systems. In *Enabling Technologies : Infrastructure for Collaborative Enterprises (WETICE), 2015 IEEE 24th International Conference on*, pages 144–149. IEEE.

[Schneider et al., 2015b] Schneider, J.-P., Champeau, J., Teodorov, C., Senn, E., and Lagadec, L. (2015b). A role language to interpret multi-formalism system of systems models. In *Systems Conference (SysCon), 2015 9th Annual IEEE International*, pages 200–205. IEEE.

[Schulz, 2017] Schulz, S. (2017). Back-in-time evaluation : Towards online trace-based debugging. In *Companion to the first International Conference on the Art, Science and Engineering of Programming*, page 40. ACM.

[Schulz and Bockisch, 2017] Schulz, S. and Bockisch, C. (2017). Redshell : Online back-in-time debugging. In *Companion to the first International Conference on the Art, Science and Engineering of Programming*, page 1. ACM.

[Seifzadeh et al., 2013] Seifzadeh, H., Abolhassani, H., and Moshkenani, M. S. (2013). A survey of dynamic software updating. *Journal of Software : Evolution and Process*, 25(5) :535–568.

[Shapiro, 1982] Shapiro, E. Y. (1982). Algorithmic program diagnosis. In *Proceedings of the 9th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pages 299–308. ACM.

[Silva, 2011] Silva, J. (2011). A survey on algorithmic debugging strategies. *Advances in engineering software*, 42(11) :976–991.

[Soria, 2011] Soria, C. C. (2011). *Dynamic evolution and reconfiguration of software architectures through aspects*. PhD thesis.

[Stallman and Pesch, 2000] Stallman, R. M. and Pesch, R. H. (2000). *Debugging with Gdb : The Gnu Source-level Debugger Fifth Edition, for Gdb Version, April 1998*. iUniverse Com.

[Steimann, 2000] Steimann, F. (2000). On the representation of roles in object-oriented and conceptual modelling. *Data & Knowledge Engineering*, 35(1) :83–106.

[Szyperski, 2003] Szyperski, C. (2003). Component technology : what, where, and how ? In *Proceedings of the 25th international conference on Software engineering*, pages 684–693. IEEE Computer Society.

[Szyperski et al., 1999] Szyperski, C., Bosch, J., and Weck, W. (1999). Component-oriented programming. In *European Conference on Object-Oriented Programming*, pages 184–192. Springer.

[Szyperski and Pfister, 1997] Szyperski, C. and Pfister, C. (1997). Workshop on component-oriented programming, summary. In *Special Issues in Object-Oriented Programming-ECOOP'96 Workshop Reader*. Heidelberg : Dpunkt Verlag.

[Taing, 2017] Taing, N. (2017). Run-time variability with roles.

[Taing et al., 2016a] Taing, N., Springer, T., Cardozo, N., and Schill, A. (2016a). A dynamic instance binding mechanism supporting run-time variability of role-based software systems. In *Companion Proceedings of the 15th International Conference on Modularity*, pages 137–142. ACM.

[Taing et al., 2017] Taing, N., Springer, T., Cardozo, N., and Schill, A. (2017). A rollback mechanism to recover from software failures in role-based adaptive software systems. In *Companion to the first International Conference on the Art, Science and Engineering of Programming*, page 11. ACM.

[Taing et al., 2016b] Taing, N., Wutzler, M., Springer, T., Cardozo, N., and Schill, A. (2016b). Consistent unanticipated adaptation for context-dependent applications. In *Proceedings of the 8th International Workshop on Context-Oriented Programming*, pages 33–38. ACM.

[Tamai, 1999] Tamai, T. (1999). Objects and roles : modeling based on the dualistic view. *Information and Software Technology*, 41(14) :1005–1010.

[Tamai et al., 2005] Tamai, T., Ubayashi, N., and Ichiyama, R. (2005). An adaptive object model with dynamic role binding. In *Software Engineering, 2005. ICSE 2005. Proceedings. 27th International Conference on*, pages 166–175. IEEE.

[Tanimoto, 2013] Tanimoto, S. L. (2013). A perspective on the evolution of live programming. In *Proceedings of the 1st International Workshop on Live Programming*, pages 31–34. IEEE Press.

[Tanter, 2004] Tanter, É. (2004). Reflection and open implementations. Technical report, Technical report, University of Chile.

[Tanter et al., 2003] Tanter, É., Noyé, J., Caromel, D., and Cointe, P. (2003). Partial behavioral reflection : Spatial and temporal selection of reification. *ACM SIGPLAN Notices*, 38(11) :27–46.

[Teitelman, 1978] Teitelman, W. (1978). *INTERLISP Reference Manual : Contrib. by JW Goodwin...* Xerox Corporation.

[Teitelman and Masinter, 1981] Teitelman, W. and Masinter, L. (1981). The interlisp programming environment. *Computer*, (4) :25–33.

[TelePharo, 2018] TelePharo (2018). Tools to manage and develop remote pharo images. `https://github.com/pharo-ide/TelePharo`. Accessed on : 2018-07-22.

[Tesone et al., 2016] Tesone, P., Polito, G., Fabresse, L., Bouraqadi, N., and Ducasse, S. (2016). Instance migration in dynamic software update. In *Meta'16*.

[Ungar and Smith, 1987] Ungar, D. and Smith, R. B. (1987). *Self : The power of simplicity*, volume 22. ACM.

[Van Cutsem and Miller, 2010] Van Cutsem, T. and Miller, M. S. (2010). Proxies : design principles for robust object-oriented intercession apis. In *ACM Sigplan Notices*, volume 45, pages 59–72. ACM.

[Vandewoude, 2007] Vandewoude, Y. (2007). Dynamically updating component-oriented systems.

[Vandewoude and Berbers, 2002] Vandewoude, Y. and Berbers, Y. (2002). Run-time evolution for embedded component-oriented systems. In *icsm*, page 0242. IEEE.

[Vandewoude and Berbers, 2004] Vandewoude, Y. and Berbers, Y. (2004). Supporting run-time evolution in seescoa. *Journal of Integrated Design and Process Science*, 8(1) :77–89.

[Vandewoude et al., 2007] Vandewoude, Y., Ebraert, P., Berbers, Y., and D'Hondt, T. (2007). Tranquility : A low disruptive alternative to quiescence for ensuring safe dynamic updates. *IEEE Transactions on Software Engineering*, 33(12) :856–868.

[Vasseur, 2004] Vasseur, A. (2004). Dynamic aop and runtime weaving for java-how does aspectwerkz address it. In *DAW : Dynamic Aspects Workshop*, volume 23.

[VisualMicro, 2018] VisualMicro (2018). Breakpoint Actions : Showing Text and Watching Expressions When a Breakpoint is Hit - Arduino IDE for Visual Studio and Atmel Studio. `http://www.visualmicro.com/page/User-Guide.aspx?doc=Working-With-Breakpoints-When-Hit.html`. Accessed on : 2018-04-27.

[Von Löwis et al., 2007] Von Löwis, M., Denker, M., and Nierstrasz, O. (2007). Context-oriented programming : beyond layers. In *Proceedings of the 2007 international conference on Dynamic languages : in conjunction with the 15th International Smalltalk Joint Conference 2007*, pages 143–156. ACM.

[Wang et al., 2014] Wang, Y., Patil, H., Pereira, C., Lueck, G., Gupta, R., and Neamtiu, I. (2014). Drdebug : Deterministic replay based cyclic debugging with dynamic slicing. In *Proceedings of annual IEEE/ACM international symposium on code generation and optimization*, page 98. ACM.

[Weyns, 2017] Weyns, D. (2017). Software engineering of self-adaptive systems : an organised tour and future challenges. *Chapter in Handbook of Software Engineering*.

[Wong et al., 2016] Wong, W. E., Gao, R., Li, Y., Abreu, R., and Wotawa, F. (2016). A survey on software fault localization. *IEEE Transactions on Software Engineering*, 42(8) :707–740.

[Würthinger et al., 2013] Würthinger, T., Wimmer, C., and Stadler, L. (2013). Unrestricted and safe dynamic code evolution for java. *Sci. Comput. Program.*, 78(5) :481–498.

[Yin, 2013] Yin, H. (2013). *Defusing the Debugging Scandal : Dedicated Debugging Technologies for Advanced Dispatching Languages*. Universiteit Twente.

[Yin et al., 2011] Yin, H., Bockisch, C., Akşit, M., De Borger, W., Lagaisse, B., and Joosen, W. (2011). Debugging scandal-the next generation.

[Zellagui et al., 2017] Zellagui, S., Tibermacine, C., Bouziane, L. H., Seriai, A.-D., and Dony, C. (2017). Refactoring object-oriented applications towards a better decoupling and instantiation unanticipation. In *SEKE : Software Engineering and Knowledge Engineering*.

[Zeller, 1999] Zeller, A. (1999). Yesterday, my program worked. today, it does not. why ? In *ACM SIGSOFT Software engineering notes*, volume 24, pages 253–267. Springer-Verlag.

[Zeller, 2009] Zeller, A. (2009). *Why programs fail : a guide to systematic debugging*. Elsevier.

[Zeller and Hildebrandt, 2002] Zeller, A. and Hildebrandt, R. (2002). Simplifying and isolating failure-inducing input. *IEEE Transactions on Software Engineering*, 28(2) :183–200.

[zeroturnaround, 2018] zeroturnaround (2018). JRebel reload code changes instantly. `https://zeroturnaround.com/software/jrebel/`. Accessed on : 2018-08-03.

UNIVERSITE BRETAGNE LOIRE / MATHSTIC

UBO
Université de Bretagne Occidentale

Titre : Adaptation non-anticipée de comportement : application au déverminage de programmes en cours d'exécution

Mots clés : Déverminage non-anticipé, adaptation de comportement, adaptation dynamique centrée objet

Résumé : Certains programmes doivent fonctionner en continu et ne peuvent pas être interrompu en cas de dysfonctionnement. C'est par exemple le cas de drones en mission, de satellites et de certains objets connectés. Pour de telles applications, le défi est d'identifier les problèmes et de les corriger pendant l'exécution du programme. De plus, dans le contexte des systèmes à objets, il peut être nécessaire d'observer et d'instrumenter individuellement le comportement de certains objets particuliers.

Dans cette thèse, nous proposons une solution d'adaptation dynamique de comportement permettant de déverminer individuellement les objets d'un programme en cours d'exécution. Cette solution est présentée sous la forme d'un patron applicable aux langages objets à typage dynamique. Ce patron permet d'implanter de façon minimale et générique des capacités additionnelles d'adaptation dynamique à granularité objet. Une mise en œuvre de ce patron pour un langage de programmation particulier permet d'instrumenter dynamiquement un programme pour collecter des objets spécifiques et d'adapter leur comportement pendant l'exécution. Nous expérimentons notre patron par des mises en œuvre en Pharo et en Python. Des déverminneurs dédiés à la mise au point de programmes en cours d'exécution sont mis en œuvre pour ces deux langages objet. Ces outils sont évalués pour des cas de déverminage concrets : pour une simulation de drones, pour des applications connectées déployées sur des systèmes cyber-physiques distants, pour un serveur de discussion en ligne ainsi que sur un défaut en production d'un logiciel de génération de documents.

Title: Unanticipated behavior adaptation: application to the debugging of running programs

Keywords: Unanticipated debugging, behavior adaptation, dynamic object-centric adaptation

Abstract: Some programs must run continuously and cannot be interrupted in the event of a malfunction. This is, for example, the case of drones, satellites and some *internet-of-things* applications. For such applications, the challenge is to identify and fix problems while the program is still running. Moreover, in the context of object-oriented systems, it may be necessary to observe and instrument the behavior of very specific objects.

In this thesis, we propose a method to adapt object behavior in a running program. This solution is presented as a pattern applicable to dynamically typed object-oriented languages. This pattern makes it possible to implement, in a minimal and generic way, additional debugging capabilities at the level of objects. An implementation of this pattern for a particular programming language makes it possible to dynamically instrument a program, by collecting specific objects and adapting their behavior during run-time. We experiment this pattern in Pharo and Python implementations with dedicated debuggers for each language. These tools are evaluated on typical debugging case studies: a simulation of drones, connected applications deployed on remote cyber-physical systems, an online discussion server and a debugging session to fix a defect in a production software.